PROFESSOR: OK, so welcome back to computational systems biology, I'm David Gifford. I'm delighted to be with you here here today. And today we're going to be talking about a topic that is central to modern high throughput biology, which is understanding how to do short read alignment, sometimes called read mapping. Now it's very important to me that you understand what I'm about to say today, and so I'm hopeful that you'll be uninhibited to raise your hand and ask questions about the fine points in today's lecture if you have any, because I'd be totally delighted to answer any questions and we have enough time today that we can spend time looking at one aspect of this problem and understand it thoroughly.

An associated topic is the question library complexity. How many people have heard of sequencing libraries before? Let's see a show of hands. OK, how many people have heard of read alignment before, read mapping? OK, great, fantastic. Let's start with what we're going to be talking about today. We're going to first begin talking about what a sequencing library is and what we mean by library complexity.

We'll then turn to what has been called a full text minute size index, sometimes called a burrows Wheeler transform index, a BWT index, an FM index, but this is at the center of most modern computational biology algorithms for processing high throughput sequencing data. And then we'll turn how to use that type of index for read alignment. So let's start now with what a sequencing library Is. Let's just say that you have a DNA sample, we'll be talking about various ways of producing said samples throughout the term.

But we're going to assume that we have a bunch of different DNA molecules. And I'll illustrate the different molecules here in different colors. And we have three different types of molecules here. Some molecules are duplicated, because as you know,

typically, we're preparing DNA from an experiment where there are many cells and we can get copies of DNA from those cells or the DNA could be amplified using PCR or some other technique. So we have this collection of molecules, and to make a library, we're going to process it.

And one of the things that we'll do when we process the library is we'll put sequencing adapters on. These are short DNA sequences that we put on to the end of the molecules to enable them to have defined sequences at the ends which permits sequencing. Now, if somebody hands you a tube of DNA like this, there are a couple questions you could ask. You could check the DNA concentration to find out how DNA is there, you could run a gel to look at the size of the fragments that you're sequencing. We'll be returning to that later, but these are typically called the insert sizes of the library that you're sequencing, the total length of the DNA excluding the adapters.

But we could also ask questions about how complex this library is, because it's possible to run experiments where you produce libraries that are not very complex, where they don't have very many different types of molecules. Now that typically is a failure of the experiment. So an important part of quality control is characterizing the library complexity where we want to figure out here complexity is equal to 3. There are three different types of molecules. And we sample these molecules.

And when we sample them, we get a bunch of DNA sequence reads. And typically the number of reads that we get is larger than the complexity of the library. Here we have a total of 12 different reads. And when we sequence a library, we're sampling from it. And so the probability that we get any one particular molecule is going to be roughly speaking equal to 1 over c, which is the complexity. And thus, we could use the binomial distribution to figure out the likelihood that we had exactly four of these type one molecules.

However, as n the number of sequencing reads grows to be very large, typical numbers are a hundred million different reads, the binomial becomes cumbersome to work with. And so we typically are going to characterize this kind of selection

process with a different kind of distribution. So one idea is to use a Poisson, where we say that the rate of sequencing is going to be n over c. And we can see that here shown on the slide above is the same process where we have the ligation of the adapters.

We have a library and we have reads coming from the library. We have a characterized library complexity here, there are four different types of molecules. And the modeling approach is that assuming that we have c different unique molecules, the probability that we'll get any one of them when we're doing the sequencing is 1 over c. And if we do end sequencing reads, we can find out the probability that we'll get a certain number of each type of molecule. Let's just stick with the first one to start, OK?

Now part of the challenge in analyzing sequencing data is that you don't see what you don't sequence. So things that actually occur 0 times in your sequencing data still may be present in the library. And what we would like to do is from the observed sequencing data, estimate the library complexity. So we have all of the sequencing data, we just don't know how many different molecules there are over here. So one way to do with this is to say that let us suppose that we make a histogram of the number of times we see distinct molecules and we're going to say that we can observe molecules that are sequenced or appear l times up through r times.

So we actually can create a version of the distribution that characterizes just a part of what we're seeing. So if we do this, we can build a Poisson model and we can estimate lambda from what we can observe. We don't get to observe things we don't see. So for sure, we know we can't observe the things that are sequenced 0 times. But for the things that are sequenced at least one time, we can build an estimate of lambda. And from that estimate of lambda, we can build an estimate of C.

So one way to look at this is that if we look at the total number of unique molecules that we sequence, which is equal to m, then the probability that we observe between l and r occurrences of a given individual sequence times c is going to be

equal to the total number of unique molecules that we observe. Another way to look at this is the very bottom equation where we note that if we look at the total complexity of the library and we multiply it by 1 minus the probability that we don't observe certain molecules, that will give an estimate of the total number of unique molecules that we do see.

And thus we can manipulate that to come up with an estimate of the complexity. Are there any questions about the details of this so far? OK, so this is a very simple model for estimating the complexity of a library based upon looking at the distribution of reads that we actually observe for quality control purposes. And let us suppose that we apply this to thousands genomes data, which is public data on human. And suppose we want to test whether this model works or not, so what we're going to do is we're going to estimate the library complexity from 10 percent of the sequencing reads, so we'll pick 10 percent of the reads of an individual at random, we'll estimate the complexity of the library, and then we'll also take all of the region the individual and estimate the complexity.

And if our estimator is pretty good, we should get about the same number, from 10 percent of the reads and from all of the reads. Will people go along with that? Think that seems reasonable? OK, so we do that. And this is what we get. And it's hard to see the diagonal line here, but there's a big oops here. And the big oops is that if we estimate the library complexity from just 10 percent of the reads, it's grossly underestimating the number of unique molecules we actually have. In fact, it's off by typically a factor of two or more.

So for some reason, even though we're examining millions of reads in this subsample, we're not getting a good estimate of the complexity of the library. Does anybody have any idea what could be going wrong here? Why is it that this very simple model that is attempting to estimate how many different molecules we have here based upon what we observe is broken? Any ideas at all? And please say your name first.

AUDIENCE: I'm Chris.

PROFESSOR: Hi Chris.

AUDIENCE: Is it because repeated sequences, so there could be a short sequence at the end of one molecule that's the beginning of another one, middle of another one, so [INAUDIBLE].

PROFESSOR: Chris, you're on the right track, OK? Because what we have assumed at the outset was that all of these molecules occur with equal probability. Right? What would happen if in fact there are four copies of this purple one and only two copies of the other molecules? Then the probability of sampling this one is going to be twice as high as the probability of sampling one of these. If there's non uniformity in the original population, that's going to mess up our model big time.

And that could happen from repeated sequences or other kinds of duplicated things, or it could be that there's unequal amplification. It might be that PCR really loves a particular molecule, right, and amplifies that one a lot, and doesn't amplify another one that's difficult to amplify. So somewhere in our experimental protocol pipeline, it could be that there's non uniformity and thus we're getting a skewness to our distribution here in our library. So the other thing that's true is in a Poisson, lambda, which is equal to the mean, is also equal to the variance.

And so our Poisson's only one knob we could turn to fit the distribution. So coming back to this, we talked about the idea that the library complexity still may be four but then there may be different numbers of molecules of each type. And here's an idea for you, right? The idea is this. Imagine that the top distributions are the number of each type of molecule that are present. And it might be that our original assumption was that it was like the very top, that typically there are two copies of each molecule in the original sequencing library, and that's a fairly tight distribution.

But it could be, in fact, that the number of molecules of each type is very dispersed. And so if we look at each one of those plots at the top, the first four, those are going to be our guesses about the distribution of the number of copies of a molecule in the original library. And we don't know what that is, right? That's something we can't directly observe, but imagine that we took that distribution and used it for lambda in

our Poisson distribution down below for sampling.

So we have one distribution over the number of each type of molecule we have and we have the Poisson for sampling from that, and we put those two together. And when we do that, we have the Poisson distribution at the top, the gamma distribution is what we'll use for representing the number of different species over here and their relative copy number. And when we actually put those together as shown, we wind up with what's called the negative binomial distribution, which is a more flexible distribution, it has two parameters.

And that negative binomial distribution can be used, once again, to estimate our library complexity. And when we do so, we have lamba be the same, but k is a new parameter. It measures sort of the variance or dispersion of this original sequencing library. And then when we fit this negative binomial distribution to that 1,000 genomes data, it's going to be hopefully better. Let's start with a smaller example. If we have a library that's artificial with a known million unique molecules and we subsample, it gives you 100,000 reads, you can see that with different dispersions here in the left, k with different values from 0.1 to 20, the Poisson begins to grossly underestimate the complexity of the library as the dispersion gets larger, whereas the negative binomial, otherwise known as the GP or gamma Poisson, does a much better job.

And furthermore, when we look at this, in the context of the thousand genomes data, you can see when we fit this how much better we are doing. Almost all those points are almost exactly on the line, which means you can take a small sampling run and figure out from that sampling run how complex your library is. And that allows us to tell something very important, which is what is the marginal value of extra sequencing. So for example, if somebody comes to you and they say, well, I ran my experiment and all I could afford was 50 million reads. Do you think I should sequence more? Is there more information in my experimental DNA preparation?

It's easy to tell now, right? Because you can actually analyze the distribution of the reads that they got and you can go back and you could estimate the marginal value

of additional sequencing. And the way you do that is you go back to the distribution that you fit this negative binomial and ask if you have r more reads, how many more unique molecules are you going to get? And the answer is that you can see that if you imagine that this is artificial data, but if you imagine that you had a complexity of 10 to the 6 molecules, the number sequencing regions is on the x-axis, the number of observed distinct molecules is on the y-axis, and as you increase the sequencing depth, you get more and more back to the library.

However, the important thing to note is that the more skewed the library is, the less benefit you get, right? So if you look at the various values of k, as k gets larger, the sort of the skewness of the library increases, and you can see that you get fewer unique molecules as you increase the sequencing depth. Now I mention this to you because it's important to think in a principled way about analyzing sequencing data. If somebody drops 200 million reads on your desk and says, can you help me with these, it's good to start with some fundamental questions, like just how complex is the original library and you think that these data are really good or not, OK?

Furthermore, this is a introduction to the idea that certain kinds of very simplistic models, like Poisson models of sequencing data can be wrong because they're not adequately taking into account the over dispersion of the original sequencing count data. OK, so that's all there is about library complexity. Let's move on now to questions of how to deal with these reads once we have them. So the fundamental challenge is this.

I hand you a genome like human. 3 times 10 to the ninth bases. This will be in fast a format, let's say. I had you reads. And this will be-- we'll have, say, 200 base pairs times 2 times 10 to the eighth different reads. And this will be in fast q format. The q means that there are-- it's like fast a except that our quality score's associated with each particular base position. And the PHRED score which is typically used for these sorts of qualities, is minus p minus 10 times log base 10 of the probability of an error.

Right, so a PHRED score of 10 means that there's a 1 in 10 chance that the bases

is an error, a PHRED score of 20 means it's one in a 100, and so forth. And then the goal is today if I give you these data on a hard drive, your job would be to produce a SAM file, a Sequence Alignment and Mapping file, which tells us where all these reads map in the genome. And more pictorially, the idea is that there are many different reasons why we want to do this mapping. So one might be to do genotyping. You and I differ in our genomes by about one base in a thousand. So if I sequence your genome and I map it back or align it to the human brain reference genome, I'm going to find differences between your genome and the human reference genome.

And you can see how this is done at the very top where we have the aligned reads and there's a G, let's say, in the sample DNA, and there's a C in the reference. But in order to figure out where the differences are, we have to take those short reads and align them to the genome. Another kind of experimental protocol uses DNA fragments that are representative of some kind of biological process. So here the DNA being produced are mapped back to the genome to look for areas of enrichment or what are sometimes called peaks.

And there we want to actually do exactly the same process, but the post processing once the alignment is complete is different. So both of these share the goal of taking hundreds of millions of short reads and aligning them to a very large genome. And you heard about Smith Waterman from Professor Berg, and as you can tell, that really isn't going to work, because its time complexity is not going to be admissible for hundreds of millions of reads.

So we need to come up with a different way of approaching this problem. So finding this alignment is really a performance bottleneck for many computational biology problems today. And we have to talk a little bit about what we mean by a good alignment, because we're going to assume, of course, fewer mismatches are better. And we're going to try and align to high quality bases as opposed to low quality bases and note that all we have in our input data are quality scores for the reads.

So we begin with an assumption that the genome is the truth and when we are

aligning, we are going to be more permissive of mismatches in read locations that have higher likelihood of being wrong. So is everybody OK with the set up so far? You understand what the problem is? Yes, all the way in the back row, my back row consultants, you're good on that? See, the back row is always the people I call on for consulting advice, right? So yeah. You're all good back there? Good, I like that, good, that's good, I like that, OK.

All right. So now I'm going to talk to you about what are the most amazing transforms I have seen. It's called the Burrows Wheeler Transform. And it is a transform that we will do to the original genome that allows us to do this look up very, very quickly. And it's worth understanding. So here's the basic idea behind the Burrows Wheeler Transform. We take the original string that we want to use as our target that we're going to look things up in, OK, so this is going to be the dictionary looking things up in and it's going to be the genome sequence.

And you can see the sequence on the left hand side, ACA, ACG, and the dollar sign represents the end of string terminator. OK. Now here's what we're going to do. We take all possible rotations of this string, OK? And we're going to sort them. And the result of sorting all the rotations is shown in the next block of characters. And you can see that the end of string character has the shortest sorting order, followed by A, C, and G and that all the strings are ordered lexically by all of their letters.

So once again, we take the original input string, we do all of the possible rotations of it, and then we sort them and wind up with this Burrows Wheeler Matrix as it's called in this slide, OK? And we take the last column of that matrix and that is the Burrows Wheeler Transform. Now yo might say, what on earth is going on here? Why would you want to take a string or even an entire genome? We actually do this on entire genomes, OK? Consider all the rotations of it, sort them, and then take the last column of that matrix. What could that be doing, OK?

Here's a bit of intuition for you. The intuition is that that Burrows Wheeler Matrix is representing all of the suffixes of t. OK, so all the red things are suffixes of t in the matrix. And when we are going to be matching a read, we're going to be matched it

from its end going towards the beginning of it, so we'll be matching suffixes of it. And I'm going to show you a very neat way of using this transform to do matching very efficiently. But before I do that, I want you to observe that it's not complicated.

All we do is we take all the possible rotations and we sort them and we come up with this transform. Yes.

AUDIENCE: What are you sorting them based on?

PROFESSOR: OK, what was your name again?

AUDIENCE: I'm Samona.

PROFESSOR: Samona. What are we sorting them based upon? We're just sorting them alphabetically.

AUDIENCE: OK.

PROFESSOR: So you can see that if dollar sign is the lowest alphabetical character, that if you consider each one a word, that they're sorted alphabetically, OK? So we have seven characters in each row and we sort them alphabetically. Or a lexically. Good question. Any other questions like that? This is a great time to ask questions, because what's going to happen is that in about the next three minutes if you lose your attention span of about 10 seconds, you're going to look up and you'll say, what just happened? Yes.

AUDIENCE: Could you explain the suffixes of t?

PROFESSOR: The suffixes of t? Sure. Let's talk about the suffixes of tr. They're all of the things that end t. So a suffix of t would be G, or CG, or ACG, or AACG, or CAACG, or the entire string t. Those are all of the endings of t. And if you look over on the right, you can see all of those suffixes in red. So one way to think about this is that it's sorting all of the suffixes of t in that matrix. Because the rotations are exposing the suffixes, right, is what's going on. Does that make sense to you? Now keep me honest here in a minute, OK, you'll help me out? Yes. Your name first?

AUDIENCE: [INAUDIBLE]. [INAUDIBLE].

PROFESSOR: [INAUDIBLE].

AUDIENCE: What is dollar sign?

PROFESSOR: Dollar sign is the end of string character which has the lowest lexical sorting order. So it's marking the end of t. That's how we know that we're at the end of t. Good question. Yes.

AUDIENCE: Can you sort them non-alphabetically, just different ways to sort them [INAUDIBLE] algorithm?

PROFESSOR: The question is, can you sort them non alphabetically. You can sort them any way as long as it's consistent, OK. But let's stick with alphabetical lexical order today. It's really simple and it's all you need. Yes. in red is the suffixes in the last colored group on the right?

PROFESSOR: No, no.

AUDIENCE: What's in red?

PROFESSOR: What's in red are all the suffixes of T on the very far left. OK?

AUDIENCE: On the right, last column group?

PROFESSOR: The right last column group. That last column in red, that is the Burrows-Wheeler Transform, read from top to bottom. OK? And I know you're looking at that and saying, how could that possibly be useful? We've taken our genome. We've shifted it all around. We've sorted it, we take this last thing. It looks like junk to me, right?

But you're going to find out that all of the information in the genome is contained in that last string in a very handy way. Hard to believe but true. Hard to believe but true. Yes. Prepare to be amazed, all right?

These are all great questions. Any other questions of this sort? . OK. So, I'm going to make a very important observation here that is going to be crucial for your

understanding. So I have reproduced the matrix down on this blackboard. What? That's usually there under that board, you know that. You guys haven't checked this classroom before, have you? No. It's always there. It's such a handy transform.

So this is the same matrix as the matrix you see on the right. I'm going to make a very, very important assertion right now. OK? The very important assertion is that if you consider that this is the first a in the last column that is the same textual occurrence in the string as the first a in the first column. And this is the second a in the last column, that's the same as the second a in the first column. And you're going to say, what does he mean by that?

OK, do the following thought experiment. Look at the matrix. OK? And in your mind, shift it left and put all the characters on the right hand side. OK? When you do that, what will happen is that these things will be used to sort the occurrences a on the right hand side.

Once again, if you shift this whole thing left and these pop over to the right, then the occurrence of these a's will be sorted by these rows from here over. But these are alphabetical. And therefore they're going to certain alphabetical order. And therefore these a's will sort in the same order here as they are over there.

So that means that when we do this rotation, that this textual occurrence of a will have the same rank in the first column and in the last column. And you can see I've annotated the various bases here with their ranks. This is the first g, the first c, the first end of line, end of string character. First a, second a, third a, second c. And correspondingly I have the same annotations over here and thus the third a here is the same lexical occurrence as the third a on the left in the string, same text occurrence.

Now I'm going to let that sink in for a second, and then when somebody asks a question, I'm going to explain it again because it's a little bit counterintuitive. But the very important thing is if we think about textual recurrences of characters in that string t and we put them in this framework, that the rank allows us to identify identical textual recurrences of a character.

Would somebody like to ask a question? Yes. Say your name and the question, please.

AUDIENCE: Dan.

PROFESSOR: Dan.

AUDIENCE: So in your original string though those won't correspond to the same order in the transformed string. So like the a's in the original string in their order, they don't correspond numerically to the transformed string.

PROFESSOR: That's correct. Is that OK? The comment was that the order in BWT, the transform is not the same as the order in the original string. And all I'm saying is that in this particular matrix form, that the order on the last column is the same as the order in the first column for a particular character. And furthermore, that these are matching textual occurrences, right?

Now if I look at a2 here, we know that c comes after it, then a, then a, and c and g, right? Right. OK, so did that answer your question that they're not exactly the same?

AUDIENCE: Yes. I don't understand how they're useful yet.

PROFESSOR: You don't understand how it's useful yet. OK. Well, maybe we better get to the useful part and then you can-- OK. So let us suppose that we want to, from this, reconstruct the original string. Does anybody have any ideas about how to do that? OK.

Let me ask a different question. If we look at this g1, right? And then this is the same textual occurrence, right? And we know that this g1 comes right before the end of character, in end of string terminator, right? So if we look at the first row, we always know what the last character was in the original string. The last character is g1, right? Fair enough? OK

Where does g1 would occur over here? Right over here, right? What's the character

before g1? c2. where is c2 over here? What's the character before c2? a3. What's the character before a3? a1. Uh, oh.

Let me just cheat a little bit here. a1 a3 c2 g1 $. So we're at a1, right? What's the character before a1? c1, right? What's the character before c1? a2. And what's the character before a2? That's the end of string. Is that the original string that we had? Magic. OK? Yes.

AUDIENCE: Wouldn't it be simpler to look at-- to just remove the dollar sign [INAUDIBLE] or do you mean reconstruct from only the transformed?

PROFESSOR: We're only using this. This is all we have. Because I actually didn't use any of these characters. I was only doing the matching so we would go to the right row. Right? I didn't use any of this. And so, but do people understand what's going on here? If anybody has any questions, now is a great time to raise your hand and say-- here we go. We have a customer. Say your name and the question, please.

AUDIENCE: My name is Eric.

PROFESSOR: Thanks, Eric.

AUDIENCE: Can you illustrate how you would do this without using any of the elements to the left of the box?

PROFESSOR: Absolutely, Eric. I'm so glad you asked that question. That's the next thing we're going to talk about. OK, but before I get to there, I want to make sure, are you comfortable doing it with all the stuff on the left hand side? You're happy about that? OK. if anyone was unhappy about that, now would be the time to say, I'm unhappy, help me. How about the details? Everybody's happy? Yes.

AUDIENCE: So, you have your original string in the first place, though, so why do you want to create another string of the same length? Like, how does this help you match your read?

PROFESSOR: How does this help you match your read? How does this help you match your read, was the question. What is was your name?

AUDIENCE: Dan.

PROFESSOR: That's right, Dan. That's a great question. I'm so glad you asked it. First we'll get to Eric's question and then we'll get to yours. Because I know if I don't give you a good answer that you're going to be very mad, right? OK?

Let's talk about the question of how to do this without the other things. So we're going to create something called the last to first function that maps a character in the last row, column, I should say, to the first column. And there is the function right there. It's called LF.

You give it a row number. The rows are zero origined. And you give it a character and it tells you what the corresponding place is in the first column. And it has two components. The first is Occ, which tells you how many characters are smaller than that character lexically. So tells you where, for example, the a's start, the c's start, or the g's start.

So in this case, for example Occ of c is 4. That is the c's start at 0, 1,2, 3, the fourth row. OK? And then count tells you the rank of c minus 1. So it's going to essentially count how many times c occurs before the c at the row you're pointing at. In this case, the answer is 1 and you add 1 and that gets you to 5, which is this row.

So this c2 maps here to c2 as we already discussed. So this LF function is a way to map from the last row to the first row. And we need to have two components. So we need to know Occ, which is very trivial to compute. There are only five elements, one for each base and one for the end of line terminator, which is actually zero. So it will only have integers and count, which is going to tell us the rank in the BWT transform and we'll talk about how to do that presently.

OK. So did that answer your question, how to do this without the rest of the matrix? Eric?

AUDIENCE: Could you show us step by step on the blackboard how you would reconstruct it?

PROFESSOR: How do we reconstruct it?

AUDIENCE: Yeah.

PROFESSOR: You mean something like this? Is this what you're suggesting?

AUDIENCE: Somehow I get a feeling that the first column doesn't help us in understanding how the algorithm work only using the last column.

PROFESSOR: OK. Your comment, Eric, is that you feel like the first column doesn't help us understand how the algorithm works, only using the last column, right? OK.

AUDIENCE: [INAUDIBLE] going back to the first column of data.

PROFESSOR: OK. Well let's compute the LF function of the character and the row for each one of these things, OK? And that might help you, all right? Because that's the central part of being able to reverse this transform. So this is, to be more clear, I'll make it more explicit. This is LF of I and BWT of i, where i goes from 0 to 6. So what is that value for this one right here? Anybody know? Well it would be Occ of g, which is 6, right? Plus count of of 6 n g, which is going to be 0.

Or I can look right over here and see that in fact it's 6, right? Because this occurrence of g1 is right here. So this LF value is, it's 6 4 0 a1 is in 1, a2 is in 2, a3 is in 3, c2 is in 5. So this is the LF function, 6 4 0 1 2 3 5. And I don't need any of this to compute it. Because it simply is equal to, going back one slide, it's equal to Occ of c plus count. So it' going to be equal to where that particular character starts on the left hand side and its rank minus 1.

And so these are the values for LF. This is what I need to be able to take this string and recompute the original string. If I can compute this, I don't need any of that. And to compute this, I need two things. I need Occ and I need count. All right? Now, I can tell you're not quite completely satisfied yet. So maybe you can ask me another question and it would be very helpful to me.

AUDIENCE: How did you get G1's last and first functions score being 6?

PROFESSOR: OK. Let's take that apart. We want to know what LF of 6 and where was that G1? G1 is 1 and 0, right? Sorry. LF of 1 and g is equal to, right? Is that g and 1 or 0? Oop, sorry it's in 0. So this is what you like me to compute, right?

OK what's Occ of g? It's how many characters are less than g in the original string? I'll give you a clue. It's 1, 2, 3, 4, 5, 6.

AUDIENCE: [INAUDIBLE].

PROFESSOR: No, it's how many characters are less than g in the original string. How many things are going to distort underneath it? Where do the g's begin in the sorted version? The g's begin in row 6. OK? So OCC of g is 6. Is that-- are you getting hung up on that point?

AUDIENCE: Yes. How do you know that without ever referencing back to the first 5 columns?

PROFESSOR: Because when we build the index we remember.

AUDIENCE: Oh, OK.

PROFESSOR: So we have to, I haven't told you this, but I need to compute, I need to remember ways to compute Occ and count really quickly. Occ is represented by four values only. They're where the a's start, the c's start, the t's start, and the g's start. That's all I need to know, four integers. OK? Are you happy with that?

Occ, the a's start at 1. The c's start at 4 and the g's start at 6. OK? That's all I need to remember. But I precompute that. OK? Remember it. Are you happy with that no?

AUDIENCE: Yes.

PROFESSOR: OK. And then this business over here of count of zero and g. Right? Which is how many g's, what's the rank of this g in the right hand side? 1. Minus 1 is 0. So that's 0. That's how we computed it. OK?

These are great questions because I think they're foundational. Yeah.

AUDIENCE: Occ and count are both precomputed as you're building on this last column [INAUDIBLE].

PROFESSOR: They are precomputed and well, I have not told you, see, you're sort of ahead of things a little bit in that I'd hoped to suspend disbelief and that I could actually build these very efficiently. But yes, they're built at the same time as the index is built. OK? But yes.

OK and if I have Occ and count and I have the string, then I can get this. But I can still need to get to Dan's question because he has been very patient over there. He wants to know how I can use this to find sequence region of genome. And he's just being so good over there. I really appreciate that, Dan. Thanks so much for that. Are you happier? OK you're good. All right.

So and this is what we just did. The walk left algorithm actually inverts the BWT by using this function to walk left and using that Python code up there, you can actually reconstruct the original string you started with. So it's just very simple. And we went through it on the board. Are there any questions about it at all?

AUDIENCE: Yes. Can you actually do this any column? Like, why are you using the last column instead of like, why can't you just change it like the [INAUDIBLE], the equation and make it work for--

PROFESSOR: Because a very important thing is that this is actually a very important property right? Which is all the suffixes are sorted here. And if we didn't do that though, I couldn't answer Dan's question. And he'd be very upset. So please be respectful of his interest here. Now, the thing is, the beauty of this is, is that I have all these suffixes sorted and what you're about to see is the most amazing thing, which is that we're going to snap our fingers and, bang, we can map 200 million reads in no time at all.

You like that? You're laughing. Oh, oh, that's not a good That's not a good sign.

Let's press ahead fearlessly, OK? And talk about how we're going to use this to map read. So we're going to figure out how to use this index and this transform to rapidly

aligned reads to a reference genome. And we're not talking about one read or 10 reads or 1 million reads. We're talking about hundreds of millions of reads. So it has be very efficient indeed, OK?

So here's the essential idea. There's the core algorithm on the slide. Which is that what we do is we take the original query that we have the read that we're trying to match and we're going to process it backwards from the end of the read forwards. And we begin by considering all possible suffixes from row zero to, in this case, it would be row seven. Which is the length of the entire transform.

And we iterate and in each iteration we consider suffixes that match the query. So in the first step, right here, let me see if I can get my point working, there we are. So in the first step here, we matching this c. OK? And we compute the LF of the top, which is this row and of the bottom, which is down here pointing off the end, and that takes us to the first d here and to this point.

Here are the two c's that could be possible matches to our query, which ends in a c. We then say, oh, the next character we have to match is an a. So we look here at the a we need to match, and starting from this row, which is row four, and this row, which is row six, we compute the LF of each one of these to figure out what rows in a precedes these c's.

And the way we compute the LF is that we use the character a to be able to figure out which rows have the a preceding the c. You can see, when we compute those LF functions, what we wind up with are these rows where we have a followed by c. So we're beginning to match the end of our read, as we go from right to left.

We then compute the same thing once again, considering the first a and ask what rows are going to allow us to put this a in front of the ac to form our entire read. And we compute the LF once again of these things. And you can see that here it takes us to this specific row aac. So that row represents a suffix that is matching our query exactly.

So we iterate this loop to be able to match a read against the index. And we're using

the LF function to do it. And it's a really beautiful algorithm. And remember, we only have the transform. We don't have the rest of this matrix.

So before I press ahead and talk about other details, I think it's important to observe a couple of things that are a little bit counterintuitive about this. One counterintuitive aspect of it is, that when I'm over here for example, and for example when I'm computing the LF here, I'm computing the LF of row two with respect to a. But there's a dollar sign there. Right?

So I'm using this to the LF function, to tell me where a suffix would be that actually follows my constraint of having to have an a be the prefix of ac, where I am right now. This code is actually not fake code. It's the actual code that's in a matcher, for matching a read against the index.

Now let me just stop right here for a second and see if there any other questions. Dan is getting now his answer to his question, right? About how you actually use this for matching reads. You do this once for every read. And it is linear time. Right? It's the length of the read itself is all the time it takes to match in a huge genome.

So once we've built the index of the genome, in fact, most of the time when you're doing this sort of mapping, you don't build the index. You download the index off of a website. And so you don't have to pay for the time to build this index. You just download the index and you take your reads and the time to match all of your sequencing reads against a certain build of whatever genome you're using is simply linear in the number of bases you have. Questions? Yes. And say your name and the question, please.

AUDIENCE: How did [INAUDIBLE] come up with intuition [INAUDIBLE]? It seems like they just pulled it out of a hat.

PROFESSOR: You know, I asked Mike that the other day. I saw him in a meeting and he sort of surprised at how this has taken off. And he told me some other interesting facts about this, which you probably could deduce. Which is that if you only want to match reads that are four long, you only have to sort this matrix by the first four characters.

But there are other little tricks you can play here.

Any other questions? Yes.

AUDIENCE: I'm Deborah. What is the FM index?

PROFESSOR: What is the FM index. Well, the guys who thought this up have the last initials of F and M, but that's not what it stands for, contrary to popular opinion. It stands for full text minute size. That's what they claim. So if you hear people talking about full text minute size indices, or FM indices, the Fm index is actually the part that was being asked over here, the Occ part and the LF part, how you actually compute those quickly.

That was what FNM contributed to this but, generically when we're talking about this style of indexing, it's called FM indexing or you might hear, I'm using a BWT. Some people will say that. But that's what FM stands for. Does that answer your question?

Excellent. OK. All right. Any-- these are all great questions. Yes.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Oh, you don't know that a and c are there, except that remember, if you look at the way that this is working, is that you're not actually reconstructing strings, you're only trying to find them. Right? And so at the end, top and bottom are going to point to the row that contains the suffix where your original read was.

And now your next question is going to be, where is that in the genome? This doesn't do me any good. I mean, the number 1 doesn't help me out here, doesn't mean anything. Not good, right? So where is it in the genome is the next question.

So we'll get to that in a second. What happens if you give me a read that doesn't match anywhere in this index? Well if you give me a read that doesn't match anywhere in this index, what happens is the top and bottom become the same. So on top and bottom become the same, it's a failed look up. All right?

And that's because the suffix doesn't exist in the index. And once top and bottom

become the same, they remain the same throughout that loop. Yes.

AUDIENCE: I'm Sally. My main question is that this doesn't provide any leeway for errors. You kind of have to be able to present all of your rates.

PROFESSOR: Sally, you're absolutely correct. I'm so glad you asked that question. Your observation is it does not provide any leeway for mismatches. And so unlike all the other algorithms we study, which had these very nice matrices and ability to assign weights to mismatches, this is only doing exact matching.

And so what you need help understanding is, how we can deal with mismatches in the presence of this. And I'll get to that in less than 10 minutes. And it won't be quite as elegant as what you saw from Professor Berg but it's what everybody does. So that's my only excuse for it OK? Yes.

AUDIENCE: What is the bottom set of arrows doing? What's its significance?

PROFESSOR: The significance of top and bottom, that's a great question. What the significance of top and bottom? Top and bottom bracket in that original matrix, the suffixes that are matching the original query. OK? And so between top and bottom minus 1 are all of the rows that have a suffix that match our original query. And if top equals bottom, there are no matching suffixes. But assuming there are matching suffixes, those are rows that contain a matching suffix. And as we progress along, top and bottom change as the suffixes change as we expand the suffix to contain more and more bases.

OK? OK. Any other questions? OK. So now back to the question over here, which is that OK, I know that we've matched, and I know that we have this hit. The question is, where is it in the Genome because the fact that it matched row one of my BWT matrix does me absolutely no good at all. Right?

Anybody have any ideas about how we could figure out where it is the genome? Given what I've told you so far, which is that you have the BWT, you have Occ, and you have count, and you can compute the LF function. Any ideas? Doesn't matter how slow it is.

OK well how could I figure out what came before aac in the genome? Yes.

AUDIENCE: So, for, like at the beginning, we rebuilt this whole string, starting at the end. You could rebuild the whole string starting at, rebuild the whole genome starting at the 1 and see--

PROFESSOR: You could rebuild the entire genome that prepends or occurs before aac, right?

AUDIENCE: Exactly.

PROFESSOR: Exactly. So that's what we can do. We could actually do our walk left algorithm. We can walk left from there, find out that we go two steps until we hit dollar sign, and therefore, the offset is two, where it occurs in the genome. So we can give a match position by walking left. Does everybody see that, that we could walk left to figure where it is? It's not fast, but it works. Yes.

AUDIENCE: I'm Ted.

PROFESSOR: Hi, Ted.

AUDIENCE: So now our function first has to take the read and it has to align it and the same, where the position is built into the end of the function, the speed of the function is now dependent on position as well. Is that right? Because the longer it takes,

PROFESSOR: I was being a little bit glib find if it matches or not this linear time. Now you're saying, hey, wait a minute. I want to know where it is in the genome. That's a big bonus, right? And so you'd like to know where that is? Yes. But I still can do that in linear time. And we'll show you how to do that in a second. This is not linear time.

This actually needs to walk back, order the length of genome for every single query. That's not good. All right? Well,

What we could do is, we could store what's called a suffix array with each row and say, where in the genome that position is. Where that row starts. And then maybe a simple look-up. That when you actually have a hit in row one, ah, OK, start your

position two of the genome.

But then the problem with that is that it actually takes a lot of space. And we want to have compact indices. So the trick is what we do is, instead of storing that entire suffix array, we store every so many rows, like every 25 rows. And all we do is, we walk left until we hit a row that actually has the value, and then we add how many times we walked left, plus the value and we know where we are in the genome.

So we can sample the suffix array, and by sampling the suffix array, we cut down our storage hugely and it's still pretty efficient. Because what we can do is, we just walk left until we hit a sample suffix array location and then add the two numbers together. All right?

So that's how it's done. OK? So that's how we actually do the alignment and figure out where things are. The one thing I haven't told you about is how to compute count efficiently. Now remember what count does. Count is a function-- but this is putting it all together where we're matching this query, we do the steps. We get the match. Then we do walk left once and then we look at the suffix to figure out where we are, right?

The business about count is that what we need to do is to figure out the rank of a particular base in a position in the transform. And one way to do that is to go backwards to the whole transform, counting how many g;s occur before this one, and that's very expensive, to compute the rank of this particular g. Remember the rank is simply the number of g's that occur before this one in the BWT. Very simple metric.

So instead of doing that, what we can do is, we can build a data structure that every once in awhile, counts how many a's, c's, g's, and t's have occurred before now in the BWT. And so we're going to sample this with these checkpoints, and then when you want to compute count at any point, you can go to the nearest checkpoint, wherever that is, and make an adjustment by counting the number of characters between you and that checkpoint. Very straightforward. All Right

So this, coming back to question, it's Time, right? --asked you need to build this checkpointing mechanism at the same time you build the index, as well as the sampling of the suffix array. So a full index consists of the transform itself, which is the genome transformed into its BWT. And they literally take the entire genome and do this.

Typically they'll put dollar signs between the chromosomes. So they'll transform the whole thing. It takes a sampling of the suffix array we just saw and it takes the checkpointing of the LF function to make a constant time. And that's what is inside of an FM index. OK?

Now it's small, which is one of the nice things, compared to things like suffix tree, suffix arrays, or even other kinds of hash structures for looking for seeds, it really is not even twice the size of the genome. So it's a very compact index that is very, very efficient. And so it's a wonderful data structure for doing what we're doing, except we have not dealt with mismatches yet, right?

And so once again, I want to put a plug in for BWA which is really a marvelous aligner. And we'll talk about tomorrow in recitation if you want to know all of the details of what it actually takes to make this work in practice. Now, it finds exactness matches quickly, but it doesn't really have any allowances for mismatches. And the way that bow tie and other aligners deal with this, and they're all pretty consistent, is in the following way, which is that they do backtracking.

Here's the idea. You try and match something or match a read and you get to a particular point in the read, and you can't go any further. Top is equal to bottom. So you know that there's no suffix in the genome that matches your query. So what do you do?

Well, what you can do is you can try all of the different bases at that position besides the one you tried to see whether it matches or not. I can see the horror coming over people. Oh, no, not backtracking, not that. But sometimes it actually works.

And just to give you order of magnitude idea about how this works in practice, when reads don't match, they limit backtracking to about 125 times in these aligners. so they try pretty hard to actually match things. And yes, it is true that even with this backtracking, it's still a great approach. And sometimes the first thing you try doesn't work, and you have to backtrack, trying multiple bases at that location until you get one that matches. And then you can proceed. OK And you eventually wind up at the alignment you see in the lower right hand corner, where you're substituting a g for an a, an a for a g, excuse me, to make it go forward.

Do people understand the essential idea of this idea of backtracking? Does anybody have any comments or questions about it? Like ew, or ideas? Yes.

AUDIENCE: What about gaps?

PROFESSOR: What about gaps? BWA, I believe, processes gaps. But gaps are much, much less likely than missed bases. The other thing is that if you're doing a sequencing library, and you have a read that actually has a gap in it, it's probably the case you have another read that doesn't. For the same sequence. So it is less important to process gaps than it is to process differences.

The reason is that differences mean that it might be a difference of an allele. In other words, it might be that your base is different than the reference genome. Indels are also possible. And there are different strategies of dealing with those. That would be a great question for Hang tomorrow about gaps. Because he can tell you in practice what they do. And we'll get into a little bit of that at the end of today's lecture. Yes. Question?

AUDIENCE: I'm Levy.

PROFESSOR: Hi, Levy.

AUDIENCE: How do you make sure when you're backtracking that you end up with the best possible match? Do you just go down the first--

PROFESSOR: The questions is how do you guarantee you wind up with the best possible match?

The short answer is that you don't. There's a longer answer, which we're about to get to, about how we try to approximate that. And what judgment you would use to get to what we would think is a practically good match. OK? But in terms of theoretically optimal, the answer is, it doesn't attempt to do that. That's a good question. Yes.

AUDIENCE: In practice, does this backtracking method at the same time as you're computing the matches or--

PROFESSOR: Yes. So what's happening is, you remember that loop where we're going around, where we were moving the top and bottom pointers. If you get to a point where they come together, then you would at that point, begin backtracking and try different bases. And if you look, I posted the BWA paper on the Stellar website. And if you look in one of the figures, the algorithm is there, And you'll actually see, if you can deconvolute what's going on, that inside the loop, it's actually doing exactly that. Yes. Question.

AUDIENCE: In practice, is the number of errors is small, would it make sense just to use [INAUDIBLE]?

PROFESSOR: We're going to get to that. I think the question was, if the number of errors is small, would it be good to actually use a different algorithm, drop into a different algorithm? So there are algorithms on FM index assisted Smith Waterman, for example. Where you get to the neighborhood by a fast technique and then you do a full search, using a more in depth principles methodology, right? And so there's some papers I have in the slides here that I referenced that do exactly that.

These are all great questions. OK. Yes.

AUDIENCE: If you're-- If you only decide to backtrack a certain number of times, like 100 times, then wouldn't like the alignment be biased towards the end of the short read?

PROFESSOR: I am so glad you asked this question. The question is, and what was your name again?

AUDIENCE: Kevin.

PROFESSOR: Kevin. Kevin asks, gee, if you're matching from the right to the left, and you're doing backtracking, isn't this going to be biased towards the right into the read, in some sense, right? Because if the right into the read doesn't match, then you're going to give up, right? In fact, what we know is the left end of the read is the better end of the read. Because sequences are done five prime to three prime and thus typically, the highest quality scores or the best quality scores are in the left hand side of the read.

So do you have any idea about how you would cope with that?

AUDIENCE: You could just reverse one of them. But you'd reverse--

PROFESSOR: Exactly what they do. They execute the entire genome and they reverse it and then they index that. And so, when they create what's called a mirror index, they just reverse the entire genome, and now you can match left to right, as opposed to right to left. Pretty cool, huh? Yeah.

So backtracking, just note that there are different alignments that can occur across different backtracking paths. And this is not optimal in any sense. And to your question about how you actually go about picking a backtracking strategy, assuming we're matching from right to left again for a moment, what you can do is, if you hit a mismatch, you backtrack to the lowest quality based position, according to PHRED scores.

We talked about PHRED scores earlier, which are shown here on the slide. And you backtrack there and you try a different base and you move forward from there. So you're assuming that the read, which is the query, which is associated quality scores, is most suspect where the quality score is the lowest. So you backtrack to the right to the leftmost lowest quality score.

Now it's a very simple approach. Right? And we talked a little bit about the idea that you don't necessarily want to match from the right side and thus, typically the parameters to algorithms like this include, how many mismatches are allowed in the

first L bases on the left end, the sum of the mismatch qualities you're going to tolerate, and so forth. And you'll find that these align yourself with a lot of switches that you can set. And you can consult with your colleagues about how to set switches, because it depends upon the particular type of data you're aligning, the length of the reads and so forth.

But suffice it to say, when you're doing this, typically we create these mirror indices that actually reverse the entire genome and then index it. So we can either match either right to left or left to right. And so for example, if you have a mirror index, and you only tolerate up to two errors, then you know that either, you're going to get the first half right in one index or the mirror index. And so you can use both indices in parallel, the forward and the reverse index of the genome, and then get pretty far into the read before you have to start backtracking.

There are all these sorts of techniques, shall we say, to actually overcome some the limitations of backtracking. Any questions about backtracking at all? Yes.

AUDIENCE: Is it trivial knowing the BWT originally to find the mirror BWT? Like for example,

PROFESSOR: No, it's not trivial.

AUDIENCE: So it's not like a simple matrix transforming [INAUDIBLE].

PROFESSOR: No. Not to my knowledge. I think you start with the original genome, you reverse it and then you compute the BWT with that. Right? That's pretty easy to do. And Hang was explaining to me today how you compute his new ways of compute BWT, which don't actually involve sorting the entire thing. There are insertion ways of computing the BWT that are very [INAUDIBLE], and you could ask him this question tomorrow if you care to come.

All right just to give you an idea on how complex things are, to build an index like this, takes, for the entire human genome, we're talking five hours of compute time to compute an index to give you an order of magnitude time for how to compute the BWT. the LF checkpoints, and the suffix array sampling. Something like that.

So it's really not too bad to compute the index of the entire genome. And to do searches, you know, we're talking about, like on a four processor machine, we're talking about maybe upwards of 100 million rads per hour to map. So if you have 200 million reads and you want to map them to a genome, or align them as it's sometimes called, it's going to take you a couple hours to do it. So this is sort of the order of magnitude of the time required to do these sorts of functions.

And there are a couple fine points I wanted to end with today. The first is we haven't talked at all about paired, erred, and read alignment. In paired read alignment, you get, for each molecule, you get two reads. One starting at the five prime end on one side, , and one starting from the five prime end on the other side. So typical read links might be 100 base pairs on the left and 100 base pairs on the right.

What is called the insert size is the total size of the molecule from five prime end to five prime end to read. And the stuff in the middle is not observed. We actually don't know what it is. And we also don't know how long it is. Now when these libraries are prepared, size selection is done, so we get a rough idea of what it should be. We can actually compute by looking at where things align on the genome, what it actually is. But we don't know absolutely.

If we were able to strictly control the length of the unobserved part, which is almost impossible to do, then we would get molecular rulers. And we would know exactly down to the base, whether or not there were indels between the left read and the right read when we did the alignment. We actually don't have that today.

The sequencing instrument actually identifies the read pairs in its output. That's the only way to do this. So when you get an output file, like a fast Q file, from a sequencing instrument, it will tell you, for a given molecule, here's the left read and here's the right read. Although left and right are really sort of misnomers because there really is no left and right, right? This is one end and then this is the other end.

Typical ways of processing these paired reads, first you align left and right reads. And they could really only be oriented with respect to a genome sequence where you say that one has a lower coordinate than the other one when you're actually

doing the alignment. And if one read fails to align uniquely, then what you can do is, you know what neighborhood you're in because you know, roughly speaking, what the insert size is, so you can do Smith Waterman to actually try and locate the other read in that neighborhood. Or you can tolerate multiply mapped reads.

One thing that I did not mention to you explicitly, is that when you match the entire query, and top and bottom are more than one away from each other, that means you've got many places in the genome that things map. And thus you may report all of those locations or I might report the first one. So that's one bit of insight into how to do a map paired reads.

And these are becoming very important because as sequencing costs go down, people are doing more and more paired and sequencing because they give you much more information about the original library you created and for certain protocols can allow you to localize events in the genome far more accurately.

Final piece of advice on considerations for read alignment. We talked about the idea that some reads will map or align uniquely to the genome and some will multimap. You know that the genome is roughly 50% repeat sequence. And thus it's likely that if you have a particular read molecule, there's a reasonable chance that it will map to multiple locations. Is there a question here? No. OK.

You have to figure out what your desired mismatch tolerance is when you're doing alignment and set the parameters to your aligner carefully, after reading the documentation thoroughly, because as you could tell, there's no beautiful matrix formulation like there is with a well established basis in the literature, rather it's more ad hoc. And you need to figure out what the desired processing is for paired reads.

So what we've talked about today is we started off talking about library complexity and the idea that when we get a bunch of reads from a sequencer, we can use that collection of reads to estimate the complexity of our original library and whether or not something went wrong in the biological processing that we were doing. Assuming it's a good set of reads, we need to figure out where they align to the genome.

So we talked about this idea of creating a full text minute size index, which involves a Burrows-Wheeler transform. And we saw how we can compute that and throw away almost everything else except for the BWT itself, the suffix array checkpoints, and the FM index checkpoints to be able to reconstruct this at a relatively modest increase in size over the genome itself and do this very, very rapid matching, albeit with more problematic matching of mismatches. And then we turned to the question of how to deal with those mismatches with backtracking and some fine points on paired end alignment.

So that is the end of today's lecture. On Tuesday of next week, we'll talk about how to actually construct a reference genome, which is a really neat thing to be able to do, take a whole bunch of reads, put the puzzle back together again. I would encourage you to make sure you understand how this indexing strategy works. Look at the slides. Feel free to ask any of us.

Thanks so much for your attention. Welcome back. Have a great weekend. We'll see you next Tuesday.