Next Generation Sequencing Technologies: Data Analysis and Applications

Read Mapping with BWT

Dr. Riddhiman Dhar, Department of Biotechnology

Indian Institute of Technology Kharagpur

Good day, everyone. Welcome to the course on Next Generation Sequencing Technology Data Analysis and Applications. In the last few classes, we have been discussing mapping algorithms, and in the last class, we talked about the Burrows-Wheeler transform-based algorithm. So, we have seen how we do the Burrows-Wheeler transform and what the properties of the transform namely the last first mapping. So, what we are going to discuss today is read mapping using that Burrows-Wheeler transform-based method, ok? So that is the agenda for today's class.

We will be looking at the process of mapping reads with these BWT-based methods. So, these are the keywords that we will come across in this class. LF mapping: we have introduced this term before. FM index is a new term that we will come to explain in this presentation and backtracking.

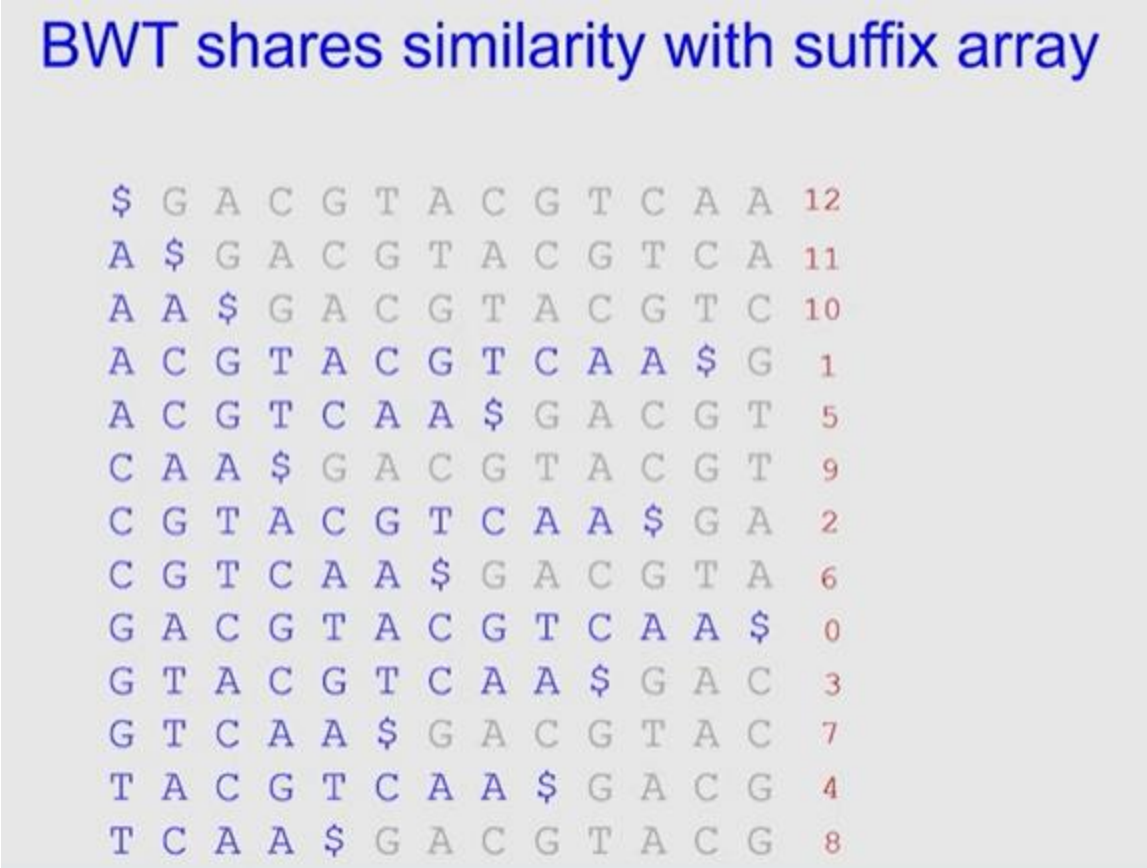
So just to summarize what we have learned regarding the Burrows-Wheeler transform, this is a string rotation coupled with lexicographical sorting of a reference sequence, and this gives rise to the Burrows-Wheeler transform. We have seen this: we start with the reference sequence, we add this dollar sign, we rotate the strings, and then we do the sorting, and finally, the last column of that matrix gives us the BWT. So, what we have also seen is that because this is just one column, BWT is of the same size as the genome sequence. So that's great.

So it means it will not take up too much space, ok? Compare this to the other methods, for example, a hash table, a suffix array, or a suffix tree, where a sequence, the reference sequence, is probably repeated multiple times. We have to store these same bases multiple times in memory. So that takes a lot of space or storage, right? So, you have seen that if we talk about numbers, then the hash table takes about 12 to 15 GB for the human genome.

For the suffix tree, it's even higher; it's between 30 and 35 GB, whereas for the suffix array, it's lower again; it's 12 to 50 GB. But still, those numbers are quite high if you want to run this on a desktop system. So hopefully this will not take too much space, and we have also talked about how once we have created this Burrows-Wheeler transformation, we can do something called last-first mapping. So, this is something we talked about in the last class, and this allows us to recreate the reference sequence from the BWT and then use the principle of LF mapping. So, once we have understood this Burrows-Wheeler transform method and last mapping, the question is: how do we utilize this BWT and LF mapping to map reads to the reference sequence?

So that's the ultimate goal of all the methods and algorithms that we have been describing. So, before we go there, I'll just quickly mention one point. If you look carefully, I will see that BWT actually shares similarities with the suffix array. So, let's take the same example of this reference sequence that we discussed in the last class. So, this is the reference; we go through the whole process that we just mentioned and discussed in the last class, and we generate this matrix, right?

And this BWT is the last column; we have the position here, ok? So, once we have this, what you can probably notice very carefully if you look at this matrix, ok, and if you first focus on these first two rows only, ok, and up to this dollar sign, ok, what you would see is that these are actually the suffixes of the reference string, right? So, if we add this dollar at the end of the reference string and then we start creating all the suffixes, you'll see that these are the suffixes that we'll get, ok? And if we extend this process, right, for the full matrix and just focus on these letters up to the dollar sign for each row, this will actually give you all the suffixes—all possible suffixes of the reference sequence, ok? So, what is also very interesting is that these are all lexicographically sorted, right?



So remember, when you store the suffix array, we actually do the lexicographical sorting. So here, Burrows-Wheeler transform, will require this lexicographical sorting, and this automatically gives you this suffix array, right? So along with the positions, you get all the suffixes, and we can actually write it in a slightly different way, right? Here you see, right? So, once we have done this, once we have written it like this, you can see, right, this actually resembles the suffix array, exactly like that, right, and this is all sorted with, and also the positions are here, ok.

So this kind of tells us perhaps we can go ahead with a similar strategy as searching a suffix array, right? So, we talked about how, because you have this lexicographical sorting, you can do a very efficient search through the matrix or through the string using something called binary search. So, the answer is actually no. So, if you are looking for the read sequence using this suffix array method, you cannot perhaps use the same strategy as in the suffix array, right? Why is that so? We do not, so suffix array representation requires storing the whole matrix, or at least up to the dollar sign.

But if you remember, in the Burrows-Wheeler transform, we do not store the matrix, right? Storing the matrix will defeat our purpose of saving space, right? So, if you store the matrix, it will take up too much space, but we do not want that. So, what we store is the last column, and we can recreate the first column out of the last column, right? So, this is the maximum amount of data that we store.

So, we cannot use the suffix array method, right? We cannot use a binary search because we do not store the full matrix that is generated in this Burrows-Wheeler transform. So, we need to think about alternative strategies, right? And this we cannot; if you go with a suffix array, then of course, it will defeat the purpose, right? We will again require a lot of storage.

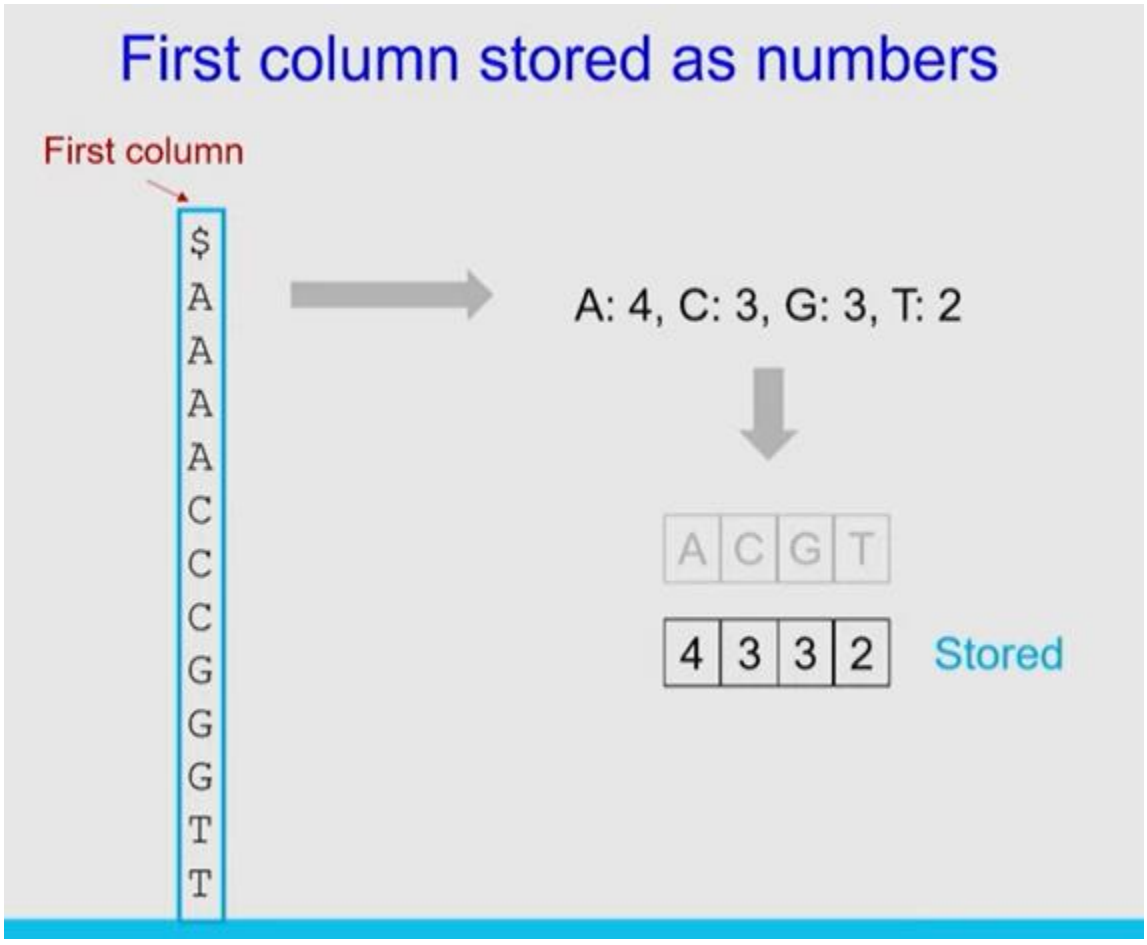
Again, the suffix array has a problem, right? You cannot map reads with mismatches, etc., right? So, we will suffer from the same problem if we just go ahead with this suffix array search method. So, how do we actually search? So, the question is, can we use this LF mapping to actually align a read to the reference sequence?

So we will see if we can do that, right? So first, let us discuss what is actually stored in this Burrows-Wheeler transform. After we have done the transformation, how is the data stored, right, and what are the values of the data stored? So, there is a concept, right? Once we have done this Burrows-Wheeler transform for a string, there is a concept called FM index that actually defines whatever data would be stored when you do this Burrows-Wheeler transform, right?

So, as you can probably imagine, we can use the Burrows-Wheeler transform to actually do string matching, pattern search, etc.; beyond this, just read mapping, right? So, actually, those are the original applications, and we have adapted this to read mapping, ok? So, this FM index is an index that contains the Burrows-Wheeler transform along with some small additional data structures, right? So, we will see what those additional ones are—at least some of those we will discuss.

The full form is full text minus paste, right, and this was proposed by Ferragina and Mangini. So, in 2000, they proposed this index. So, what we store is BWT—the transformation that we have done, right? We can also see that if you want to store the first column, right, you can generate the BWT, the first column from the BWT, but in case you want to store the first column, you can actually simply store them as numbers of letters of each type, right. So let us look at this, right, how it actually looks like, what we mean actually, right?

So if you look at this full matrix again, you have the first column and the last column and the letters in between are never stored. So, these are just shown for our understanding, okay? So, if you see the first column, right, you have these letters: A, C, G, and D. These are lexicographically sorted. So, what we can do is simply store these as numbers of A's, numbers of C's, numbers of G's, and numbers of D's, right? So, you have four A's, right, and then followed by three C's, then you have three G's and two T's, okay?



So you can simply say, like, you can count this: how many A's you have, how many C's you have, how many G's you have, and how many T's you have, and then you can simply store this as 4, 3, 3, 2, right? You do not even need to store A, C, G, and T because we know we will have these four letters, right? So, if you have these 4, 3, 3, 2, we can immediately derive, okay, we have four A's, we have three C's, we have three G's, and we have two T's, right? So, if we can come up with this, this is very efficient, right? storing just four numbers, right? So, whatever those numbers are, you do not need to store all these letters, etc.

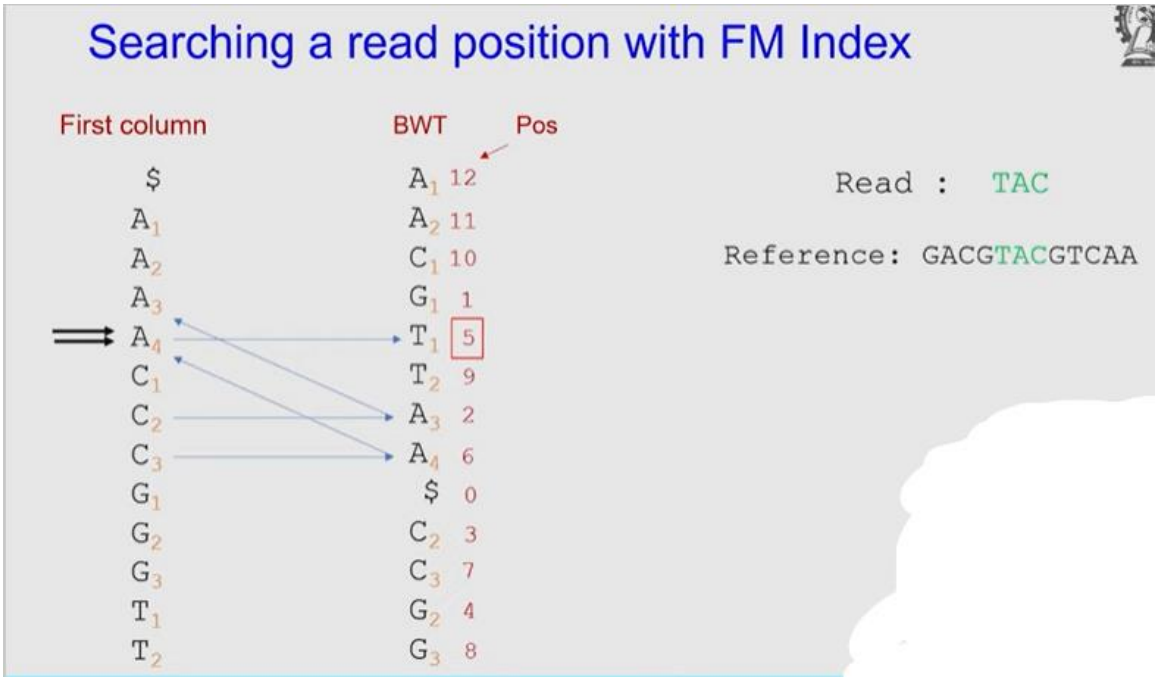
how they appear, okay? You can recreate this first column very, very easily if you just have this 4, 3, 3, 2, or something like that, right? So, you can look at another example, right, for better understanding. So, we have, let us say we are given this 3, 4, 2, 3, that is how the first column is stored, and the assumption is we have this A, C, G, T, these are sorted, right, lexicographically, and we can recreate the first column from these numbers, okay? So, what it means is we have three A's, four C's, two G's, and three T's, okay, and we can simply write starting with the dollar, we can write this three A's, four C's, two G's, and three T's, right? So, you can see that this is a very easy way to store data, and then you can generate the first column if we know these numbers very easily, okay?

So how do you now search for a read position with this FM index? Right, that index that is there has been created, okay? So, what you have again, let us go back to this BWT along with the position, right, and you have the first column, right. The first column will be created from those numbers or from the BWT, and we can probably apply the left mapping principle to actually find this readout, right? Map this read, and we will take this example. For example, we want to map this read TAC, right? We want to find the location of this read in the reference sequence, okay? Using the information and these indexes that we have created, okay? So, what are the steps? So, this actually happens in these three steps, right? So, we start with the last letter, okay?

So this happens in reverse. So, if you remember, like we are doing this for the suffix tree, we start with the first letter and then go on, right? Here it is actually the other way, right?

We start with the last letter, okay? So, in this example with TAC, we start with C, okay? So, we start first looking for C, then A, then T, okay? So it happens in reverse, and there is a reason we will see, like why that actually works, and then we use two pointers to find the range of matches, okay?

So there are two pointers; we will show them with arrows in the next example. And these two pointers actually kind of narrow down the range where the match might occur, okay? So, when you start, they are kind of taking the full matrix, right, and then as you go along, right, they will narrow down to certain choices based on the matches with the letters in the read, right. So, this is how the whole process will work, and this will finally give you the result: it will converge to probably one, right, one row, or maybe two, three rows if there are multiple matches, okay? So, let us have a look at how this is actually done, as I have mentioned.



So we start with C here, right? So, if you look at the initial state, these two arrows are the pointers, right? So, they are pointing to the first column and the last row, right? So, you have this first row, last row, so they are taking the full data. Right, full data; all the rows are there. Now, what would be the purpose of the next step? It is actually to narrow down, right? So, what we see is C there, right? The letter C happens, right? And then the pointers now kind of reduce the choice, right?

So it narrows down to the exact matches, right? So, it finds, okay, only these rows; they start with C, right? So, you have C here, so these two pointers can kind of exclude the rest of the rows now, okay? So, we can simply focus on these rows, right? So, they start with C, which means we have found our first match for the first base, right?

Now what we will use is to remember the string rotation, okay, that we have done to generate this Burrows-Wheeler transform, and as we have discussed before, string rotation means the letters in the last column, right, corresponding to these letters; they represent the letters that occur before this letter in the original string, right. So, let us take an example that will be clear, right? So, if you see here that this is C and the corresponding letter in the last column is T, right? So, what it means is that in the reference string, this T comes before this C, okay, because of the string rotations, okay. Similarly, if you look at these other two rows here, right, you have C, and then you have A3 here, C2, and A3.

What it means is that this A3 occurs before C in the original string and in the reference string, okay? So, we will remember that, right? So, what we can then say is, okay, we were searching for A, okay? So only these two, C2 and C3, have A before Cs, okay? So, what this would mean is that we are narrowing down our choices further, right?

We are kind of narrowing down to choices where we see A before C, right? So because in the read, we have A before C, we narrow down our choices to those rows where we see A occurring before C, right? So this is the next step. Again, once we have done this, we will use the principle of last-first mapping, right? So that is why I have written them as A1, A2, A3, etc.

so that we can easily map them back to the first column, right? So now we will use the LF mapping, right, and these A3, A4, correspond to A3, A4 here. So this A is in the first column, okay? What is the next step? The next step is to search for T, right? So look for the letters that occur before these As, right, before A3 and A4.

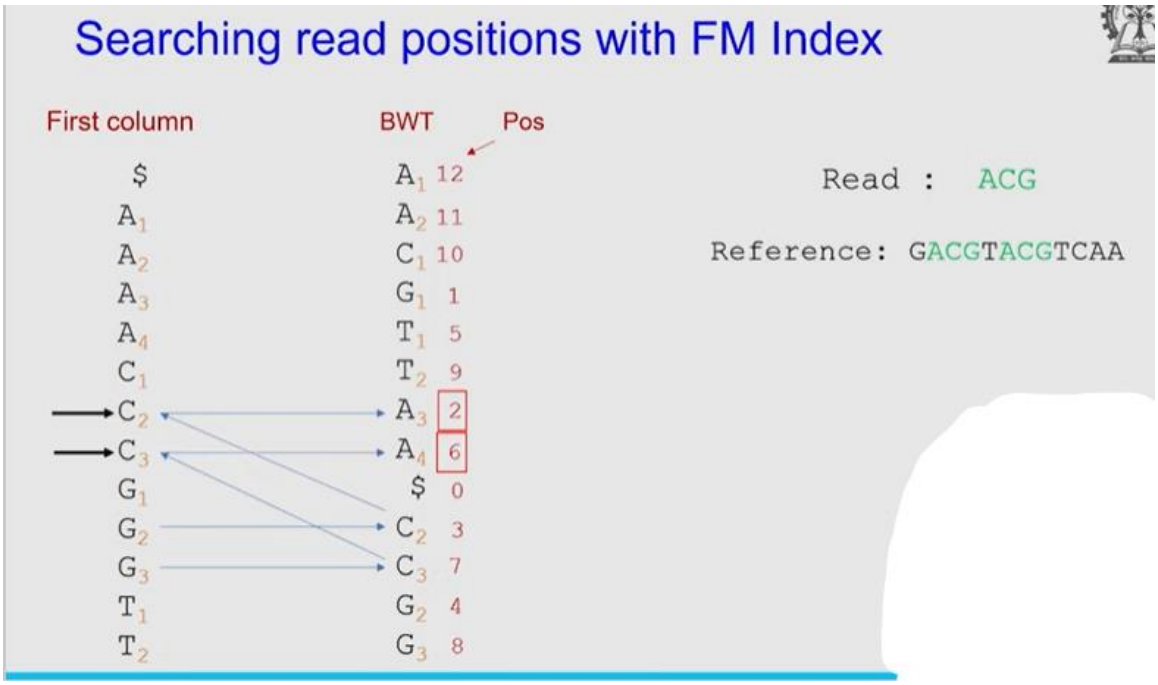
What is the letter that occurs before A3? G1, right, but G is not the match, right? So we want to find T. What about A4, right? So A4, we have the letter T1, so T before this A, which means we have now narrowed it down to the single choice, right? We have just one choice. We have completed the mapping, and we have found the read in the reference data, right?

So what you have seen now is that we go through this last first mapping, right, and we also remember the string rotation in your mind, okay? So once we have this, once we combine those two ideas, we can map this very, very easily. And of course, since we have the positions along with this BWT, we know the location of this read in the reference sequence, okay? So here is the reference sequence for your information, right? As you can see, it occurs in the fifth position, right.

So if you start counting from 1 to 1, right? Here G is 1, then this read occurs in the fifth position of this reference sequence, okay? So let us take another example, which I think would illustrate this in a better way. So we again start with these pointers that are pointing to the first row and the last row, right? So we want to find this read ACG in the reference data, okay?

We start, right? We have discussed what we do. So we start with the last letter, which is G here, okay, and we narrow down our choices, right? So again, the pointers move, and this kind of finding G is only in this position. So then you have, like, a smaller choice now, right? There are only three rows that will be considered. So we have found these first three spaces, right?

So we have found G. Now we use this again; remember the rotation, right? So we want to see what the letter is that occurs before this G, okay? So what we observe here, right, is that for G2, C2 occurs before G2, right? So there is a C, and for G3, there is a C also, right? So both of these choices are valid, right? Because we have found the second letter C in both of these choices, okay?



So what we will have to do now is continue our last first mapping, right? So we are in the last column, so we will again map back to the first column, which means this will correspond to C2 and C3 in the first column, right? What is the next step? We want to find A, right? And what we do again is look for the letter that occurs before G. This C2 and C3 and see which one would actually match this A, whether there is any A at all, right? So what we do is look at the rotation, right? We keep the rotation in mind again, and we see both C2 and C3. They actually give A before this C, right?

So you have A occurring before C in both cases, right? So both C2 and C3 have A in front of them, right? So they will be matched, okay? So now what it means is that we have found two positions in the reference sequence that actually match the read, okay? So this is again a read mapping to repetitive sequences, and this can happen, as we have mentioned, for example, in human genomic contents with a lot of repeats, so this will be quite a frequent occurrence, right? So from what you have seen, there was no issue at all finding the repetitive sequences, right?

So it just tracked all the paths again using LF mapping and keeping the rotation in mind; it found all the maps very, very easily, right? It did not take much time, et cetera; it just had to find these paths and keep the locations, okay? So I hope this clears up the concept of

mapping read positions with this FM index, the data that is stored, and as you can see, we do not read the whole matrix data, right? So we do not need to store the full matrix data; just the last column and first column in a very compact form can give us this mapping very efficiently, right? So this is a very quick process, as you can see; you do not have to go through a lot of elaborate steps again.

All right, so let us take another example, and that will probably help us understand how we actually traverse through mismatches, okay? So here is the ATC; we follow the same steps, right? We start with these two pointers; we start with the last letter C; we have narrowed down our choices; these three Cs are there, right, and we see what letter occurs before each of these Cs, right. So what do we observe? So we have T before C, right?

So this matches the second letter in the text, right? So you can see here that we have found Tc. Then we use the last mapping, right? So here is the last first mapping, and we found T2, and what we see is that G occurs before this T, right? So, but there is no G in our read, right? So the pointers are lost immediately if there is no exact match, okay?

So this again could be a problem, right? So how do you solve this? How do you actually work around this problem? So what we have seen is that if you have a perfect match, right?

If a read matches perfectly with your reference data, it can find that location very efficiently and very quickly, right? There is no issue. But if there is a mismatch, again, the pointers will be lost, right? So what do you do in that case? So this is the question that we are left with now: how do you search for reads with mismatches using this FM index? Can you do something? And it turns out that you can, and this method is called backtracking, okay?

And it allows for mismatches, right? And you can also set a mismatch threshold, right? How many mismatches would you allow, right, in your mapping process, okay? All right, so let us look at this example now slightly bigger read GAAC, right, using the same first column and same BWT, okay? So you have now learned this process very well, right? So we have three Cs, and we look at the letter that occurs before Cs, right?

So we have As and there are two matches now, and we use the last mapping, the pointers. Here you can see the pointers now moving, right, and we have this A3, A4, right, in the first column, and then we look for the letter that occurs before this As, right, and what we find is G or T. But in the read, we do not have any G or T; right, we have A. So, what this backtracking does is allow for this mismatch. It says, Okay if you do not find any exact match, let us go ahead with whatever option we have,"

So the options are G and T, okay? So, what it will do is just follow; it will allow this G and T, right? So, it will not kind of stop the process; it will say, okay, let us allow G and T, right? So, we will allow G and T, right, and we will continue the alignment, right? So, what will happen now again with the last first mapping? So, we see G1 here and T1 mapping to the first column, right, and we continue this process, and we see this T1, right; the letter before this is G, right?

So this G again matches with the letter in the read. So, what we have found is that instead of this G A C, we have actually found this G T A C, right? So, if you allow for these mismatches, if you allow for this backtracking, it will actually find some mapping somewhere, right, and here it found, right, that we have G T A C, right, that is the best match we have, but there is one mismatch, right. So, one of the issues that you can have if you have a lot of flexibility, right? If you allow backtracking, what can happen is that it will probably take all possible options in all steps. So, you can see this, right, in this graph, right, in this process, that if you allow these mismatches, the program may not stop, right, anywhere. It might allow a lot of these mismatches, and you may end up with excessive backtracking. So, to prevent that, right, there are ways to prevent this, and it is called double indexing.

So there is this index that is created from the forward sequence, so it is called the forward index, and then you have indexing from the backward side, right? So, it is not a reverse complement; it is a reverse index. So, you will see a backward index also, right? And then there are other issues here, right? So how do you actually store, so as I said, we store this B W T, right, the last column, right, we have this later.

How do you store the rank of these letters? So, we need to know this A 1, A 2, C 1, C 2, right, to do the L-F mapping, right? So how do you store the rank of these letters in the last column? If you want to store them like just by storing everything, imagine a human genome; it will take up a lot of memory. And if you are storing this BWT on the hard drive, it will also end up taking a lot of storage, which will make the whole process a lot slower. So, what we can do is perhaps store only at certain positions, right? So, if what I am talking

about is these numbers 1, 2, right, which one is the first day, which one is the second day, and which one is the third day, right, you need to keep track of that information for doing the last first mapping, right?

So if you have to store that, right, how can you store that without storing everything? And what some programs have come up with is like just store at certain locations, right? So, the store information is only for certain locations; these ranks are okay. So instead of storing for all positions, maybe store this information every 5 bases or 10 bases, right? You could vary this, and that will reduce the storage requirement a lot, right?

So this is what is done, and this is called checkpoints, ok? So, these checkpoints are the points or locations where these counts are stored, right? The ranks are stored for these letters, right? So, if you are, for example, mapping, in this kind of situation, if you are mapping and you want to find the rank of a, let us say one letter, you will not find the rank of that letter stored. So, what you can do is go to the nearest checkpoint, right, and see what the rank of that checkpoint is, and then you can deduce the rank of that position. So, in that way, you reduce the memory requirement or the storage requirement, ok? So, what are the advantages of BWT-based mapping? So, we have learned how we can actually map reads and the reads that match perfectly, as well as reads that contain mismatches.

So what are the advantages? So, what we have seen is that this is a very fast process; you do not have really complex steps, etcetera, or time-consuming steps. So, this is a very fast process. The memory or storage required is much less compared to other methods that we have discussed so far, ok?

So in comparison, this method takes about 1.5 GB less than the human genome. So, considering all FM index, double index, checkpoints, etc., you can work with only 1.5 GB of memory or RAM, right? So, it means it is now possible to run on your next computer, right? And we have seen that if you allow for backtracking, you can map reads with mismatches, and these mismatches could be mutations or sequencing errors. So, you want to keep them; you do not want to discard these reads, and you can also set thresholds for

how many mismatches you would allow in your mapping process.

And we have also seen that if you have read that map to multiple positions or repeat sequences, this method can actually handle them without any problem, right? So, it will find all those positions, and it will report these multiple positions in the final results. So here are the references that we have discussed in this presentation. So, to finally conclude, we have defined something called the FM index, which can store all necessary data in a very efficient manner. We have seen that the memory requirements for this process for storing this FM index are actually much lower than the other methods that we have discussed, for example, the hash table or the suffix array-based methods.

So suffix arrays and hash tables take about 12 to 15 GB of memory for storing human data, right? Here, you can get away with only 1.5 GB, right? So, it is almost 10 times less, right?

So that means you can now run this mapping on your desktop computer quite easily, ok? So what we have also seen, right, is that if we use this FM index and the principle of LF mapping, we can map reads very quickly, right, without any problem, and we can also map reads containing mismatches by using a method called backtracking. So combined, this seems to be the ideal method, right? So before we started the discussion on the Burrows-Wheeler transform, we started with a wish list, right? So, we wanted a method that does not take too much memory, a method that is fast, then a method that can handle reads with mismatches, and a method that can map reads to repeat sequences, right?

So it seems like this method has checked all the boxes, right? So that is why this is the method of choice, and for most of this and most of the mapping that we do in NGS datasets, we use this method most of the time. And there we have it. Thank you.