Next Generation Sequencing Technologies: Data Analysis and Applications
Burrows-Wheeler Transform (BWT)

Dr. Riddhiman Dhar, Department of Biotechnology
Indian Institute of Technology Kharagpur

Good day, everyone. Welcome to the course on Generation Sequencing Technologies, Data Analysis, and Applications. We have been discussing over the last few classes the algorithms that are used for read mapping, ok? So, we have discussed why this is a very challenging problem. We have a huge number of reads, we have a big genome, and we need to find the location of those reads in the reference genome. So, this process of combining all these factors starts to take a lot of time, right? So, that is why we need specialized algorithms, and we have discussed two such algorithms so far.

We have talked about hash table-based algorithms, and we talked in the last class about suffix tree and suffix array-based mapping algorithms. So, in this class, we will be talking about Burrows-Wheeler transform-based algorithms. So, the Burrows-Wheeler transform is a specific transform of the string or the reference sequence, and we use that transform to develop mapping algorithms, and we map reads using those that transform. So, this is what we will be discussing in this class: Burrows-Wheeler transform-based mapping.

We will discuss what this Burrows-Wheeler transform is, and then we will discuss the mapping algorithm. So, these are the keywords that we will come across in this presentation: one is string rotation, another is lexicographical sorting, and the third is left mapping. So, we will discuss these concepts in detail, but you will see these terms throughout the presentation. So, just to briefly recap the algorithms and mapping methods that we have discussed so far, ok? So, we have talked about hash table-based mapping methods, suffix tree-based methods, and suffix array-based methods, right?

So, hash table-based method, right? So, we talked about how we stored the reference sequence in a hash table, right? We generate c's and store them in a hash table along with the locations, and we generate similar c's from the read data and search in the hash table to find the locations. So, of course, there are a lot more details in there. And then we talked about suffix tree and suffix array-based methods; these are related ideas.

So, they all relate to suffixes, right? And we talked about how we generate suffixes from a reference sequence, and then we talked about the tree representation and the array representation, right? And we talked about these advantages, disadvantages, and how we can search sequences using the suffix tree or suffix array structure. And to summarize, we have seen that hash table and suffix tree-based methods are really fast, right? So, we can map it very quickly, and if you compare it with a blast or brute force event, these are incredibly fast, ok? But the problem is that they require a large amount of memory or storage, right?

So, all these methods take at least 12 to 15 GB of RAM if you are working with the human genome, which means you probably would not be able to run this on the desktop system. So, the suffix tree, for example, takes even more; right now, it takes about 30 to 35 GB of RAM if you are working with the human genome, right? Because of that elaborate data structure. So, the other disadvantage is that building this structure takes quite a bit of time. So, you have to build this hash table, which means you have to extract those seeds, represent them as keys, and then store the values. All that takes time. Similarly, building the suffix tree structure also takes quite a bit of time.

So, the time is there that you have to give for at least generating this data structure. In the case of a suffix tree or array, once you generate this data structure, you can store it; you do not have to build it again and again. Whereas, for the hash table, if you change the seed size to k, you will have to rebuild the hash table. Again, the k length can change depending on the sequencing platform that you are using, the read length that you are using, and the sequencing error that we expect in the data. What we have also seen is that hash table-based methods are poor at mapping reads to repeat regions.

So, you have it will take more time because you have to do this chain and alignment or chain extend, right for different positions, right multiple positions, and that increases time. On the other hand, suffix tree and suffix array-based methods can make reads to repeat regions very quickly. So, we have this because of the data structure; this is possible, but these are very poor at mapping reads with mutations or sequencing errors, right? So, this is something we have seen because they will not be able to find matches in the data structure. So, they will return; right now, there is no match.

So, this is not suitable, right? If you are discarding a read because there is a mismatch, then we will probably discard a large fraction of the data. So, this is a way to do better, right? So, what we have seen is that these two algorithms can do some of these tasks pretty well, but we probably need a more efficient algorithm. What we actually need is kind of like the wish list, right?

So, we have an algorithm that does not take up too much memory or storage, just like the earlier ones, ok? So, we can run it on our computer, on the desktop, or even on the laptop, right? It should be fast and scalable. So, if you are dealing with billions of pieces of data, it should not be a problem; you should be able to complete the mapping within a reasonable amount of time. And then, of course, this method should be able to handle reads with mutations and sequencing errors because this is part and parcel of mutation sequencing data, right?

So, you will have reads with sequencing errors and mutations, and it should also be able to map reads to repeat regions quickly, right? It should not be taking too much time to do this process, ok? So, can we design this kind of algorithm, or is there an algorithm that can do that? So, what researchers looked at is something called the Barrows-Wheeler transform, ok? So, this is a transform applied to strings, and this idea was taken from there and applied to the read mapping algorithm.

So, what we will discuss now is this Burrows-Wheeler transform-based mapping method. So, first we will talk about Burrows-Wheeler transformation, or, in short BWT. So, what are the steps? We will first describe the steps, and of course, we will take examples and do them, and we will implement them and see how they actually work. So, the first step is to add a dollar sign to the end of the string.

So, if you are taking a reference, a genomic reference sequence, we add this dollar to the end, ok? And with the assumption that the dollar is lexicographically smaller than all letters of the alphabet, right? So, especially if you have ATGC, etcetera, the dollar is smaller than that. So, when you sort, the right dollar will come first, right? So, it kind of hints at the fact that there would probably be a sorting step later on, right?

Then what we do is something called string rotations, ok? So, we generate all possible string rotations from the reference sequence, ok? So, we will see how this works in a moment with an example. Once we have generated these string rotations of the original reference sequence, right, what we do is something called sorting, ok? So, we sort the original string and its rotations lexicographically, ok?

So, we have seen lexicographically sorting just in the case of a suffix array. So, here is a similar idea, right? We generate all these rotations, take the original string, and sort lexicographically. And this matrix that we generate, the last column of that matrix, is the Burrow-Wheeler transform of the original string, ok? And then this matrix, this Burrows-Wheeler transform, has certain properties that you can utilize for the mapping, ok?

So, we will slowly elaborate on this whole process, starting with the Burrows-Wheeler transformation, ok? So, let us take an example, ok? So, here is the reference sequence we have taken, ok? So, we have this basis. So, step 1 is in dollars, right?

So, that is what we have done here, right? We have added this dollar at the end, ok? The next step is the rotation of the strings, ok? So, for the first rotation, right, we are rotating. Imagine you are rotating in a clockwise or anti-clockwise direction, right, so in one direction, right. So, how do you rotate the string? How do you visualize this process? So, imagine you are, right? You have written these letters on a cylinder, right?

So, you have a pointer at a specific position in the cylinder, and you read the letter in that position, right? That is the start of the string, and the rest of the letters are around this cylinder, right? You are rotating this cylinder, and you are just visualizing the start position. Of course, the start string and start letter will change, and the string will rotate. So, this is the process; that is what we are doing, ok? So, you can visualize this. We are bringing the dollar first, and then we are shifting the rest towards the end, right?

So, then we have this dollar first, then we have GCCG, TCA, and GTCA, and we continue this process until we reach going back to the original string, ok? So, let us do all these rotations carefully. Have a look at what we are actually doing. So, what is the next step, then? So, we will bring this last a to the first position, right and then shift the rest of them to the end, ok? So, this is the process, and you can carefully see it, right? So, what we have done is we have done these rotations, ok?

So, here you will see that we have actually kind of taken the dollar first, right? This is first here, yeah, and then we have got this a here from the beginning, and then we have pushed the rest of the m towards the end. Then we have put this here, right? So, we now have two a's in the beginning and have pushed the rest towards the end, right?

So, this process continues, right? So, then we get the C; this C comes here; then we get the t here; and so on, right, and we continue until we reach this original string, right. So, this is the last string, right? Because if you put g in the front, what we will get is this one, right? We will get back to the original string, ok? So, we continue this rotation and generate all possible rotations, ok? So, this is step 2, and this gives us the whole matrix, ok?

So, this is the matrix that we generate through this string rotation, ok? So, once we have generated these rotations, we take the original string and the rotations, and we do the sorting. Here we go back to the sorting step. Go to the sorting now step, right? So, please have a look carefully, right?

Which one would be the first one? Which would be the first here, right? Once we sort all these strings, which one will be the first string that we pick? So, the first one would be this one, right? Remember we mentioned the dollar is smaller than the rest of the letters?

So, this will come first, ok? So, what do we do? This will be the first. So, we bring it to the first position, ok? So, this is the first one, the first string. Then, from the rest, we have to choose the second one now, ok?

So, the second one is again will. So, it should start with a, I think, right. So, we have four letters: A, T, G, and C. So, if you sort that lexicographically, we have A, C, G, and T. So, all the strings starting with a will come now, ok? Now, out of those strings that start with a, you have 1, 2, 3, 4, and which one would come first, ok?

So, if the first letter is the same, we look at the second letter, ok? And if we look at the second letter, we see this one has a dollar, right? Again, remember that the dollar is smaller than all the letters. So, this one should come now, ok? So, this is the one that should now be taken out, right?

So, we will move this here; we will take this here after. So, this will form the second string, right? So, this is the first; this is the second, right? So, this is the sorted part; the first two are now sorted. Now, we have to sort with the rest of the matrix, ok?

So, for the third position, what do we have? We have again these ones with a's, right, and the first letter is the same for all of them; a is the first letter, but for this one second letter is a, for this one second letter is c, and the second letter is c here, right? So, surely this one will now have to be moved here, right? So, this will move, ok? So, this one is what we have chosen, and this will move here, all right.

So, this is the third one. Now, this is the sorted part, right? So, we continue, right, and for this one, we have now this one here. So, again, we have two strings that start with a, and the second letter is also the same for these two; the right C, the third letter is also the same, and the right fourth letter is also the same. So, we keep on, right? If they are the same, right? Keep on going, right? C, G, T, and fifth letter: we have now resolved this, right? So, in the fifth letter, this one has a, the other one has C, ok.

So, this one, the one with a will, will come, right? So, this will now be brought up here. So, this will come here, so this part is now sorted, ok, and we have to continue this process. We have one more with A, so this one will undoubtedly appear now, right this will in the if you sort this, right this will appear here, ok. So, this is getting sorted out slowly. Now, we have exhausted all the strings that start with A, right?

So, we need to look at strings that start with C, right? Again, there are three, right, that start with c, then we look at the second letter: we have A, we have G, we have G, ok. So, this one starts with A; the second letter is a, right? So, we will choose this one, ok, and this will come up here, ok, and we have sorted this one. We have two more with C's, right and again, if you do the same comparison, you will see this one is in the fourth position and this one has C, right.

So, this one will get priority, of course, right? The one with a and this will be brought here, ok? So, this is the sorted part now, and we have just one more remaining with c. So, this will come up, and we have generated the sort. Now, out of these five that are remaining, we have strings that start with g, and some strings start with t, right? Again, the g will get preference, and again, we have a very similar order, right?

So, you see G and A, G T, G T, right? So, here you have GTA, and here you have GTC. So, you can now guess the order, right? So, this one will come first, then this one, then this one, right? So, we can actually go through this quickly, right? So, this one will be first; this is then this one, right; it is taken up; and this one is here, right.

So, this is now sorted, and then finally, these two are right. So, you have t c a and t a c; it is easy, right? So, this one will get preference; you are in the second position, right? So, this will come up, and this is the final one, and we have generated the sorted list, ok? So, this is the sorted matrix that we have for all the strings and that we have generated by rotations, ok? So, in this matrix, we will now perform all sorts of tricks, and we will see how this can help us map those things, all right?

So, once we have done this process, we have done these three steps, ok? This is the Burrows-Wheeler transformation of the original string, ok? So, through these steps, we have generated this Burrows-Wheeler transformation, and we simply write this, right? So, this is the original string or original reference sequence, and we have generated the Burrows-Wheeler transformation, which is the last column of this matrix. So, what you see and what you probably notice is that this structure, this transform, actually generates a string that is the same size as the reference sequence, right?

So, it is just a plus one. So, you have a dollar here extra, but this is roughly the same size as the reference sequence of the reference genome, right? So, which means it would probably not take too much space in the memory, right? So, that is a very important consideration for us, right? So, earlier methods that we have talked about all took a lot of memory, and we could not run them on desktop computers for that reason. Here it seems the memory requirement will be less, and perhaps we will be able to run this index computer.

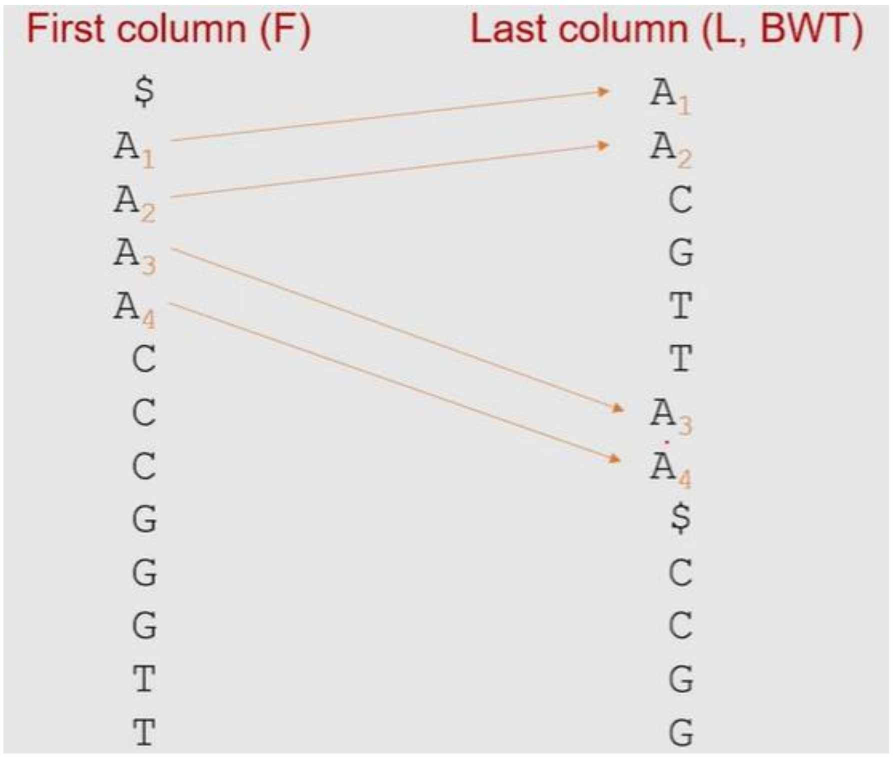
We will see in a moment, ok? So, this is the Burrows-Wheeler transformation of the original string, and we will utilize this information to actually do the mapping, ok? So, it turns out there are certain properties of this transform, ok? So, BWT actually has certain properties that we can utilize to do read mapping, ok? So, we can also add positions. We can keep track of these positions for each of these letters, and we can store these positions.

Again, this will not take too much memory or storage space if we have to store this position. So, what we have discussed at BWT size is the length of the reference, right? Now, what is actually stored in the memory? So, we can actually simply store this BWT, right, the last column, and it turns out you can actually generate the first column of that matrix from this BWT information alone, ok? So, if you just simply store this information, if you store this column in memory or in a file, right, because once you have generated this BWT for a reference sequence or reference genome, you can store it, right, and every time you have to do mapping, you can simply load that information into memory, ok. And it turns out that if you store this BWT, you can actually generate this first column, ok, because the first column is lexicographically sorted. Right, just by sorting this BWT, you can generate the first column, and you can probably match this against the matrix, ok.

So, this information is very helpful because if you just store this information here along with positions in a file, when you are running this program, you can generate the first column. Now, why do we need the first column? So, the BWT, remember, is the last column, and this is the first column of this matrix. And it turns out there is a property called last first mapping, or LF mapping. So, this property of BWT is very useful, and we can utilize this property to actually do all the red mapping, ok?

So, what do I mean by this last first map mapping? So, here is the first column I have written by F, and the last column is written by L, which is also the Burrows-Wheeler transform, or BWT. So, it says the last first mapping is that the ith occurrence of character C in the last column and the ith occurrence of character C in the first column both correspond to the same character in the reference sequence, ok? So, they are actually the same character in the reference sequence. If you count the fourth occurrence of a character, let us say the base A in the last column and the fourth occurrence of the character base A in the first column, they correspond to the same base in the reference sequence, ok? So, what it means now is that if we let us say that number A is in the first column by A 1, A 2, A 3, A 4, right, we are distinguishing this 3, 4 A's by these numbers or the subscripts, ok?

What it means is that these 4 A's, right, they actually correspond to these 4 A's here, ok, in the last column, ok. So, this correspondence is shown by these red arrows, ok? And if that is the case, if they correspond to the same base in the reference sequence, we can also number these A's by A 1, A 2, A 3, and a 4. So, this is what we mean by last first mapping, ok? So, in this order of appearance of these letters, they remain the same in the first column as well as in the last column, ok?



And we can now extend these to all bases, right, and we can number these occurrences of C, G, and T by these numbers, the subscripts C 1, C 2, C 3, G 1, G 2, G 3, and T 1, T 2, right, and they will again correspond to the bases or the order of occurrence in the last column, ok? So, this is the last first mapping, ok? So, this is the property that is very helpful for read mapping, ok? So, we will see this, right, how we can actually utilize this information to actually do read mapping. Now, we will talk about another property: what we can do with LF mapping, ok?

So, using this BWT sequence and the property of LF mapping, we can actually generate the original reference sequence, ok? The only thing we need is this BWT, and we can generate the first column from BWT. Combining these two with the principle of LF mapping, we can go back to the reference sequence, ok? So, I will illustrate that in a moment, right? So, here is the first column, right, and we have the last column, which is the BWT, ok?

And remember, we can generate the first column from BWT. We have generated this, and what we will do is do this last mapping, or something called traversing. So, we will start with this first letter in the last column, ok, and we will see, right, we will map this to the corresponding a, right. So, this 1 corresponds to this one here, right, because of the LF mapping property, ok? Now, this is a 1. What is the letter that is before this a?

Remember this string rotation we are doing in such a way, right? So, we are bringing these letters; maybe you can have a loop, right, that the matrix, and that will make things clear, right? So, when we are doing this string rotation, let us say here that what it means is that if you look at this string, this is the letter that is before this a, right? This a is before this a here in the reference sequence. Similarly, if you take this example for this string, this T occurs just before this C in the reference sequence, ok?

So, the string rotation actually means this, right? So, we can actually map the letter that is before this letter in the first column, ok? So, this is something we are actually getting here, right? So, let us go back there, ok? And here we have a 1 that corresponds to this one. Now, because of this property of this string rotation, what it means is that this a, right, occurs before this a, right.

So, A 1 and A 2 occur before a 1 in the reference sequence, right? So, we have now written this reference. We are writing this reference sequence, right? So, we have a, we have a, right? So, we have written down two a's, ok? Now, similarly, this C occurs before this a, right?

So, we can do this mapping again, right? So, what it means is that we have now added C, right, because this C occurs before this a, ok? And then this C for this C T, this T is mapped, right? This T occurs before this C; we have added this T now for that in the reference sequence that we are building, right? And for this T, we can see see this G will be before this T, right?

And we have added G now to the reference sequence. And then we again use the last first mapping here, right. So, we this G 3 corresponds to this G 3 here, right. And we can continue this process, we can see this C is before this C because of the rotation. Again, we go back here to the first column using last first mapping. We look at the letter that is before this C which is a, right, and do last first mapping so on, right.

We can repeat this process right through these columns—right, first column, last column—until we reach the dollar sign here, right. That would be the N, ok? And this is the reference sequence that we generated, right? So, hopefully it is clear, right, how we can utilize this information of BWT and the first column using L F mapping to generate the original reference sequence, ok?

So, it means this is a very efficient way of storing the genome information, right? Just storing the BWT can give you the reference genome sequence; it can also help in read mapping, ok? So, we will see how we can utilize this information and the last mapping property to map reads in the next class, ok? So, this is the next class, right? We will utilize this BWT information and L-F mapping to map reads to the reference sequence, ok?

These are the references that we have used. So, what we have seen is that we can generate this Burrows-Wheeler transform through string rotations and lexicographical sorting. We have talked about three steps, starting from the reference sequence. One is the addition of this dollar sign, and the dollar is lexicographically smaller than all the letters in the alphabet. And then we do the string rotations, right?

We generate all possible rotations until we reach the original string, right? We come back to the original string, right? And in the third step, we combine the original string with all the rotations to do something called lexicographical sorting, right? And we have seen the process of lexicographical sorting and how we make those decisions for sorting, right? If the first letter is the same, we look at the second letter.

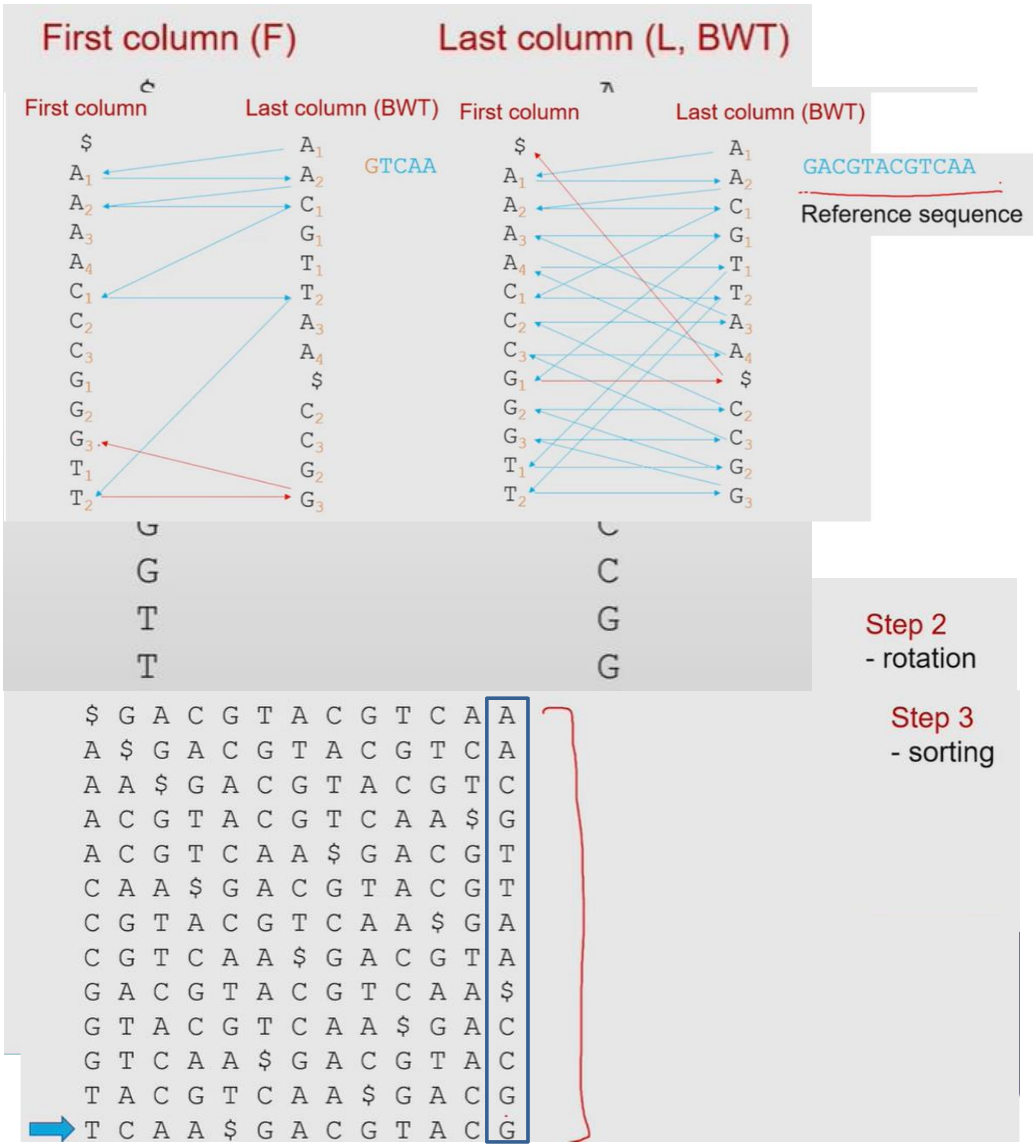
If the second letter is the same, we look at the third letter, and so on, right? And then we sort these strings, and we generate this sorted matrix. Now, once we have generated this sorted matrix, what we have seen is that the Burrows-Wheeler transform takes the same space as the genome. So, it is a very efficient way to store the genome data, right? It is not taking a huge amount of space like a hash table, a suffix array, or a suffix tree, right?

So, that is the good news, right? We can probably run to the next step. We will see in the next class, right, whether that is possible, how much we will quantify the time and the memory requirement for read mapping, ok? And what we have seen is that this Burrows-Wheeler transform actually has very interesting properties, right? So, we can generate the first column of the matrix from BWT, right? So, if you take the BWT, you can simply sort lexicographically, right, and that will give the first column, right.

And we have seen a property called last first mapping, right? And utilizing this BWT with the first column and the elect mapping, right, we can generate the reference sequence, ok? So, we will keep this in mind, and in the next class, we will see how we can utilize this information, this BWT, the LF mapping, etcetera, to actually map reads against reference sequences in an efficient manner. And we will also look at whether we can handle reads that contain mismatches or sequencing errors. That is a very important requirement for us.

And we will also see how this method will deal with repeat mapping, right? So, multiple mappings of a read will happen in the case of reads mapping to repeat sequences in the genome. This is also a very important requirement, right? So, because the human genome or other big genomes contain many repetitive elements, So, there are many repeat regions, and you will get a lot of reads that will map to these repeat regions. And if this read mapping to repeat regions takes a lot of time, that will actually increase the overall mapping time, right?

So, these are the properties that we will need for a good mapping algorithm and to develop a good mapping tool, ok? So, in the next class, we will continue discussion to see how we can map reads using this Burrows-Wheeler transformation and whether we can address those issues that we have seen earlier. That is it for today. Thank you.



```
$ G A C G T A C G T C A | A
A $ G A C G T A C G T C | A
A A $ G A C G T A C G T | C
A C G T A C G T C A A $ | G
A C G T C A A $ G A C G | T
C A A $ G A C G T A C G | T
C G T A C G T C A A $ G | A
C G T C A A $ G A C G T | A
G A C G T A C G T C A A | $
G T A C G T C A A $ G A | C
G T C A A $ G A C G T A | C
T A C G T C A A $ G A C | G
T C A A $ G A C G T A C | G
```