Next Generation Sequencing Technologies: Data Analysis and Applications
Suffix tree-based mapping algorithm

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

Good day, everyone. Welcome to the course on Innovation Sequencing Technologies, Data Analysis, and Applications. We have started discussing the mapping algorithms, and we have talked about blast, blast, etcetera, and their limitations. In the last class, we talked about the hash table-based mapping algorithms, and we have seen that they are really fast, but they require a lot of memory, and they also have problems with mapping repeat sequences to repeat sequences. So, in this class, we will be talking about the suffix tree-based mapping algorithm, and we will talk about how this algorithm can circumvent some of the issues of the hash table. So, these are the things on the agenda for this class.

So, we will talk about the suffix tree-based mapping algorithm, and then we will talk about the suffix array-based mapping algorithm. So, we will talk about what a suffix is, what a suffix tree is, what a suffix array is, etcetera, and then we will see how we can map reads to the reference sequence using this method. So, these are the keywords we will come across: suffix tree and array. So, just to briefly recap from the last class, we have talked about the hash table-based mapping algorithm, and we have seen that this is a very fast mapping process because we are storing this hash table in memory and we can search very efficiently.

Now, hash tables require a lot of memory, right? So, we have talked about 12 to 15 GB of space for storing, for example, a human genome hash table and what we have also seen is that the speed of mapping is reduced if we are mapping reads to repeat sequences. So, if you have multiple mappings across that, you know the speed is reduced because you need to extend this seed and stitching process for multiple regions. So, we have this, which actually takes more time. So, is there any alternative strategy? So, one of the things that researchers have come up with is something like a data structure that can handle mapping to repeat sequences.

So, at least we can address the issue of this mapping to repeat sequences, okay? These are the structure suffix trees and suffix arrays, and these have been used in pattern searching data mining, etcetera beforehand. So, how does this work, ok? And we will now discuss this slowly, step by step, and we will build these suffix trees and then suffix arrays and we will see how we can search very efficiently using these data structures. So, what we do is convert the reference sequence into a suffix tree. We will see in a moment how we do that, and what we will also see is that repeat sequences are represented by overlapping paths in that tree. So, which means mapping reads to repeat sequences becomes very efficient, and then we search these to the suffix tree again, we will discuss this in much more detail as we go along. So, to start with, what is a suffix? So, a suffix is a substring of a string that starts at any position in the string but ends at the end of the string.

So, let us take some examples, and then I think it will be clear. So, here is a string right; it is shown here on your screen: A C G A C C A G G A T C, and here are some examples of suffixes. So, we have G A T C, or the one below, and you can see they can start at any position in the string, but they should end at the end of the string, right? So, it should end with this, right? So, it ends here at the end of the string, ok?

So, that is why these are suffixes, but the one example below here is not a suffix because it does not end at the end of the string. So, it ends at the T, which is not the end of the string, ok? So, I hope this is now clear: what is a suffix to a string? So, we can take a very quick example, and we can write down the suffixes of this string. So, we have this here.

So, we can know what the suffixes are. So, if we start with g, we have C G A C G, G A C G, and so on. So, you can finally write all of them down, the T T A G C G A C G right. So, these are all the possible suffixes of the string OK. So, this concept will be useful when you actually go for suffix tree construction.

Now, the question is: What is a suffix tree? We understand what a suffix of a string is, but what is a suffix tree? So, this is a data structure, and it contains something like what we call the root. So, it will have one root and it will have leaves, ok, and this is actually how this data structure is built from the letters of a string. So, let us go into that. So, what these trees contain are paths from the root to the leaves.

So, in a moment, we will see the structures and what they actually look like, and we will understand this better. These paths correspond to the suffixes that exist in the string. So, we have seen all possible suffixes, and we will have paths corresponding to each of these suffixes, and all suffixes of the string will be represented by a path that will join the root and go to a leaf in that tree, ok? So, this will be the suffix tree, where all the suffixes of a string are represented. Now, as you can probably understand, what we do is take the reference genome sequence and build the suffix tree out of it.

Thus, the size of the tree is proportional to the size of the genome. If you have a bigger genome, the suffix tree will be larger, and the time taken to build the tree is also proportional to the size of the genome. As you can imagine, for bigger genomes, you will have a larger number of suffixes, and it will take more time to build the suffix tree. So, this is a kind of structure that you will see when you build a suffix tree. You will have a root here, and as you can see, this is the root, and you will have leaves. These are the leaves that you can see; they are marked with red arrows. So, these are all leaves, and you have parts from the root to the leaves, ok, and all these parts represent the suffixes of a string, ok. So, now let us take an example. Let us take a reference sequence and let us build the suffix tree of that reference sequence, and then we will understand this better.

So, how do you construct a suffix tree? We will see this process in a moment. So, let us take this reference sequence, and these are all the suffixes that are possible right from this reference sequence. I have just added a hash at the end because it denotes the end of the string. So, we can have them as the leaves, and that will say this is the end of the path that we cannot go beyond, ok? So, we will take all these suffixes and build the suffix tree, ok?

So, we start with this hash right; we start with the root; we have the base; then we have the hash right; and we have reached the leaf; ok, this is the end of this suffix; ok, end of the string. And then we take the next one, which is the g-a hash right; this is again highlighted in green, and we add this from like a root to the leaf right. So, again, it ends with this hash here, ok? So, we have g and a in between, and we can continue this process. We can do this for g and a we can do this for g and a. So, one of the things you will probably notice right

So, for this one right C G A, we have added this now G C G A, where the first letter is g right. So, g already exists, and there is a path from the root to g. So, we will use that path, and then we go and see right whether there is any c after the g. So, there is none. So, we have to construct a new path to get to the leaf, and we can continue this process and generate the full suffix tree.

Now, we can also add the positions right, which will denote where the suffix starts in the string. So, if we say this is position 1 right, then we can add this position to the leaf right saying. So, which will denote OK? This suffix starts at this position. So, I will just write the positions here, so you can see if you kind of tally with this. So, you will see that this is the position for each of these suffixes.

So, this position actually helps us in finding the location. If we are searching now for each in the reference sequence using this suffix tree, we know the location, ok, where this suffix actually starts. So, we will take another example and do the suffix tree construction. This is a slightly more complex example, and as we will see when we go and build the suffix tree, So, as before, we now have a larger number of suffixes; we need to consider all of them, and we need to find paths that go from root to leaf for each of these suffixes. So, we will see right; we will go one step from there, but I will explain, ok? So, again, we do this step by step.

So, we have the first c right, and then you have this path here from root to c, and then we have this hash right. So, this suffix is now represented in the tree. We take the next one right root; this path already exists. So, and from there, we take this c and hash right. So, we have reached the leaf.

So, this suffix is now also represented. The next thing we have is GCC, right? So, what we have is g right from root to g, then to c right, and then c here and then hash right. This path represents this suffix. Then we have C G C C, right? So, one thing we have seen right now is that this root-to-c path already exists; we will utilize that, but there is no root-to-g path here.

So, we will actually have to add that here at that node, and then we will add this g to c to hash right. So, we have reached the leaf, and this represents this suffix. Then we have this: Now, what we will see right from this root to g to c is that this path already exists, and what we need to do is add a path to g right and then to c to c, and then we have reached the leaf, ok? So, this way, we can go on, right? For all of them, we can do this, and we will end up with a tree structure like this, ok?

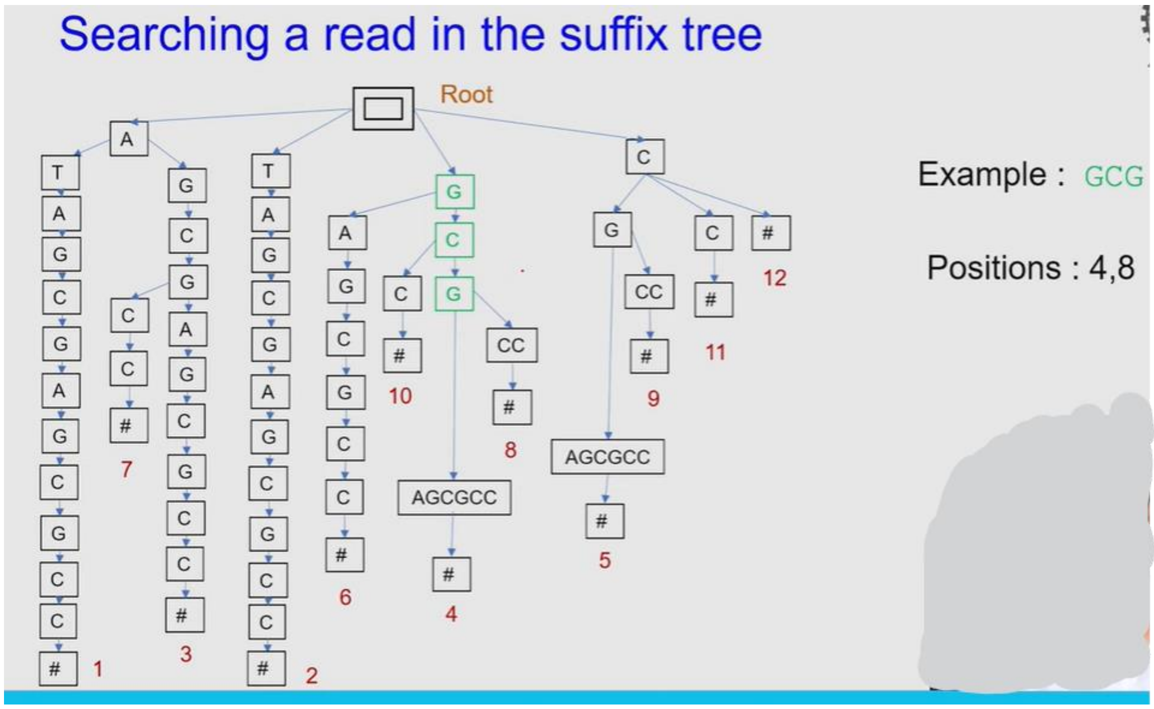
Now, you have to do it by hand. So, I will encourage you to take any example like this and do it yourself, and then you will understand the construction process. What you will see is that the first step is to identify all the suffixes of a string and then generate this suffix tree structure, keeping this simple principle in mind. So, you should have a path that connects the root to the leaf and represents each of these suffixes. So, for each of these suffixes, you will have a path in the tree, ok, and as you can see, we will see that we have all the paths here, ok.

So, I hope this is clear. Now we can also add the positions again. If we consider this to be position 1, we can again add all these positions in the tree, and that will help us identify the location of the reed when you are doing the mapping. So, what we will now do is also see how we can compress the tree a little bit, and then we will see how we search for reeds using this suffix tree structure. So, the first thing is that we can compress the tree rather than this really elaborate display right of the whole thing we can actually compress right. So, for example, where we have just one path, here we have only this one path, right? So, we can actually compress this into just double C, and we can see that this will look much simpler, and similarly, for this path, this is a single path.

So, we can just compress it right into this kind of thing. So, we can do this. So, I have done only for part of them; you can do for the whole tree like this, right? This becomes much clearer when you look at it, ok? So, we will go back to the original tree, and we will see how we can actually search for these reeds using this suffix tree structure. So, that is what we are going to do. The steps are: we start at the root, you start matching letters from the read, and you continue till the full reed sequence has been found.

So, this is very simple, right? We start at the root, go on one by one, and start matching letters, ok? So, let us take this example of GAG, and we will try to identify this GAG here. So, here we have right, we have this G, ok, you can see this green base right. So, we start from the root, and the first base we look at is G, and you see, from the root to G, there is a path.

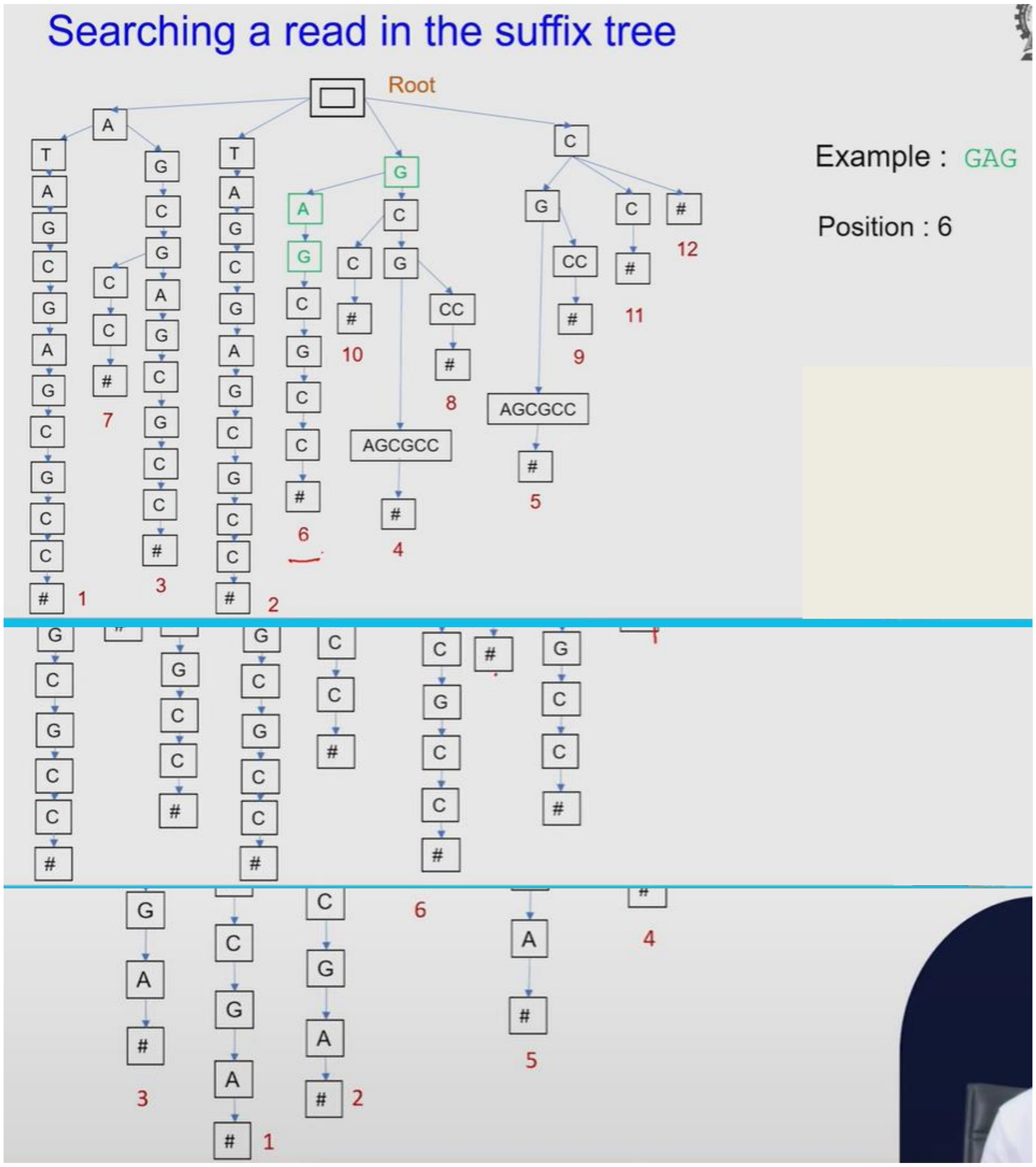
So, we are taking this. So, we have identified G. The next base is A right, and this is where we have the match right GA. The last base is G, and we have identified this G. So, we have identified or found the full string in the suffix tree. So, this is the GAG, and once we have found it, we can also look at the location right. The position is 6 positions right, and if you go back to the origin string, you will see that this GAG is at the 6 positions of the reference genome. So, it sounds very simple, right? The search process becomes very simple, and we will now take another example, and this is a slightly interesting example. We will see in a moment why that is the case. So, this example is GCG. We are now searching for GCG in the tree. Ok, we start at the root; the first letter is G, and we have found this path here: G.



The next one is C, and we have this G to C path here and then the last letter is G, and we have found G OK. So, you see, we have found this GCG here. Okay, what about the location? So, one of the things you see is that this is part of two paths, right? So, we have one path going in this direction and another path going in this direction, and they are both located in two different positions in the reference string, right? What it means is that this sequence is repeated in the reference sequence, where when you construct the tree, they actually overlap with each other because of the structure, right? So, you can then identify that there are two positions where you find where we find this ring matching, ah, position 4 and position 8, because we have two paths that are going through these ah nodes, right?

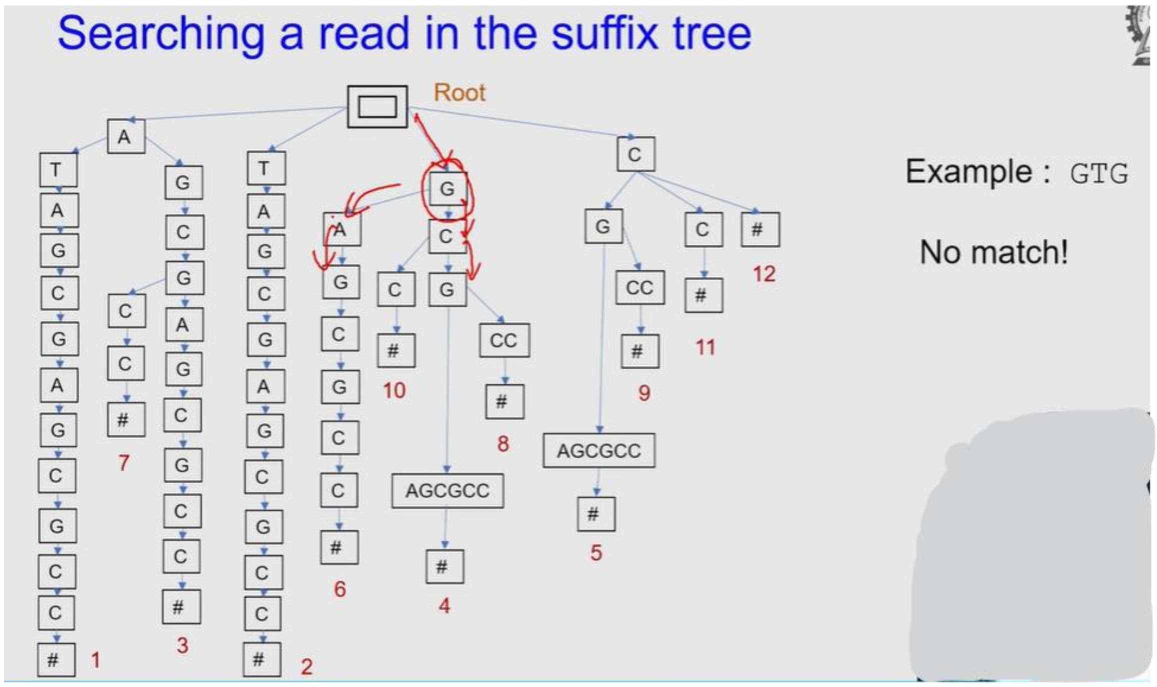
So, GCG is okay. So, what you see is that searching for repeat sequences is actually very easy in this kind of structure, right? So, you do not have to worry about this repeating this process multiple times across multiple paths, like we do in hash table-based algorithms. So, we will take another example, which is GTG, and we will repeat the same process. So, we will start with G ah again, taking from the root, and we come up to this point, right? But then we are stuck because there is no T ah G to T path, right? So, the next letter is T, and there is no path from G to T.

So, there is no match in that ring, ok? So, this is something that you probably realise is a limitation of this kind of structure, right? So, it cannot actually identify any mapping if there is a mismatch in the read data. Now, you might say that maybe we can circumvent this in some way; maybe we can allow for certain mismatches and then see if there are other matches later on, etcetera. So, those can be done right. So, for example, you can think that if you allow this, there is no T.


Searching a read in the suffix tree

Example : GAG

Position : 6



So, maybe we can allow both of these paths and we can say okay, there are A and C, but after that, we have G here, and we also have G here, and these could be the matches, right? So, you can allow for mismatches. Now, this is something that can be done, but researchers have not invested so much time and energy because there are other limitations of the suffix tree. So, which we will discuss now. So, what are the advantages of suffix tree-based methods? So, what we have seen is that we can find matches very quickly. We have seen some very simple examples, and we can find multiple mappings or mappings to repeat sequences quite easily.

It is very quick because of the overly specific data structure that you are using. The path actually overlaps in the tree, but there are major drawbacks, and that is why we have not invested that much time or energy into developing this further. So, of course, it requires a large amount of memory or storage when you are generating this tree structure. So, it has been seen that it takes about 30 to 45 GB of space for the human genome. So, this is something that you cannot run on an ordinary AH laptop or desktop; you need specialised AH servers for doing this work. And another thing is the building of the suffix tree. It is a very elaborate and complex data structure, and it is quite time-consuming, but you can argue that once you have built this for a specific reference sequence, you can store it and use it, but then the first drawback comes in because it takes a lot of storage to get and keep it on your hard drive.

And what you have seen at the last is that it is also poor at handling mutations or sequencing errors. Right when there are mismatches, it cannot find those mappings, but of course, as we have discussed, you can perhaps design certain ways to circumvent this problem, but because of the first two limitations, it is actually not worth it. So, can we reduce the amount of memory that we are using? Is there any way that you can actually reduce the memory usage because this is a major limitation? So, can we address some of these tropics? The first tropic is especially right. So, it turns out that there is a way right, and it is through the use of suffix arrays. So, what are suffix arrays? So, this is again a set of suffixes in the genome that are sorted lexicographically.

So, we will see again the examples, and we will see how we actually build these suffix arrays. So, here is a reference string again, right, and we have all the possible suffixes of that reference string, and we are also again denoting the end of the string by this hash sign, right, and we have written down all the suffixes here. So, if you remember the definition that we just mentioned, a suffix array requires lexicographical sorting or alphabetical sorting. So, we sort these ah suffixes right lexicographically, and we also have the start positions right. We can map the start position of each of these ah suffixes along with the ah suffix sequences right, and then we can do a lexicographical sorting. So, this sorting works in this way. If you are not familiar with lexicographical sorting, we first look at the first letter.

So, let us see. So, here, among all these suffixes, we have this one that comes first, right? So, so these suffixes will come at the top right; this will occur first right. So, lexicographical order is: a will come first, then it will be c, then it will be g, then it will be t. So, we start with that. So, we start with these suffixes, which have a in the first place. If the first letter is the same, then we look at the second letter.

So, in this case, this one is g, this one is g here, and this one is t, which means this one will come after the right one. So, because it comes after g, Now that the second letter is also the same for these two suffixes, we look at the third letter. So, they are also the same. We look at the fourth letter, then they are also the same g and g, and then we look at the fifth letter, and finally, we see that this one has a and that one has c. So, which means lexicographically, this one would be the first one, this would be the second one and so on, ok?

So, this is how we do the sorting, and you will see that this one will come first: this C C G A G C G A right because of this occurrence of a, then we have the other one that we have talked about, then we have the a t one right, and you understand why that is the case. We repeat this process for all the suffixes, right? So, then, after a, you have the suffixes that start with c right, as you can see here, and then we have the suffixes that start with g right, and finally, the suffix that starts with t. And again, the principle is the same: if we look at the first letter, if it is the same, we look at the second letter, if it is the same, we look at the third letter, and so on, and then finally, we sort them lexicographically. Right somewhere, it will give you this difference, and you can sort lexicographically. Now, once you sort this lexicographically, the sorted start positions also appear right. So, you are also sorting the start positions and you get these sorted start positions.

Now, what is the point of this sorting? Right, why do you do this sorting at all? Now, what happens is that if you want to search for reads, it turns out that you can actually sort the search for reads very efficiently through this sorted data structure. So, here are some points before we actually go to the search part, right? So, what we store are the suffixes; we do not store them; we only store the sorted start positions. Why is that the case? Because we can actually recover the suffix from each start position and the full reference genome sequence. So, if we know the start position, we can say that this is the suffix that we are working with.

So, that is actually a more efficient way of storing things. So, we only store the positions and the reference genome sequence we have, and we can generate the suffixes whenever required. So, how do you actually search through these suffixes? So, we have these sorted suffix positions, right? So, we have this suffix array, and this is the array, which is only the positions, and we do not store these suffixes in the data. Now, how do you actually search this read sequence? So, we have, this is a multi-step process, as we have only discussed the first step.

So, we actually generate the suffixes from the position. So, once we have stored the suffix array and the genome sequence, we can generate the suffixes from the positions. We compare with read sequence, as we will see in how you actually compare, and we do lexicographical sorting, as you have seen, and that makes the search process very efficient. So, again, let us take an example, and that will be much easier to understand, ok? So, let us take this read, GCG, okay? We have taken this example in the case of the suffix tree also, and let us see how we can find this GCG algorithm.

So, how do you actually search? So, what happens is that we can use some sort of search algorithm, something called a binary search. So, you can actually break down this array into two parts and then see whether this GCG will be in this part on the other part right, and if you see the other part, you then again break it down into two sets and search through that. So, we are not going into the details of those algorithms, the search algorithms, but it will be enough to know that because of the sorting that we have done, the lexicographical sorting, we can use a very efficient search process like binary search, which allows us to find this very quickly. So, what we do is then look for this, and we find that these are the suffixes that start with this GCG. And I have not discussed the search process itself, but because of the sorting, it is a very efficient process.

So, once we have found this, we can look at the locations. Right now, you have 4 and 8, and this is where this read actually occurs. So, if you remember, this read occurs at two places. This maps to multiple regions, and here we see them because they are mapping to two different positions, right, two different suffixes, starting from two different positions. So, this means this is a repeat sequence as present in the reference genome. So, what are the advantages and drawbacks of suffix arrays? So, they take so. The first advantage is that they take less space compared to suffix trees, and it is about 12 to 15 GB for the human genome. Again, this is comparable to the hash table.

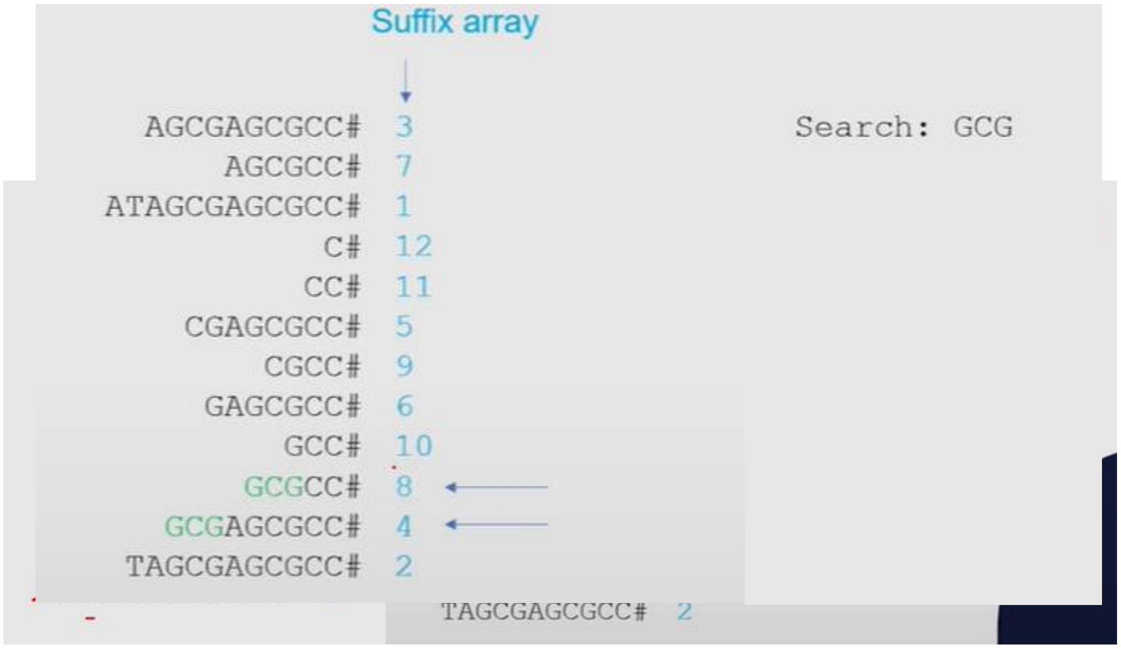
And one of the things that is also an advantage of this method compared to a hash table is that it can map reads to multiple positions or repeat sequences very easily again because of the lexicographical sorting. But the problem remains right: the major drawback is that you cannot map reads with mutations or sequencing errors; it is very poor at this kind of mapping. Again, we can think about some way of overcoming this problem, but then again, one of the issues we have seen is that the memory requirement is quite large, and I mean, this is the reason why we started looking for other algorithms compared to hash tables. Because of the memory requirement, this kind of memory may not be available on normal desktops or laptops. So, this is one tool that actually uses this suffix array method, and I encourage you to go through the reference, and you can probably see how they are utilising the suffix array-based method for searching reads.

So, here are the references that we have used for this class. To summarise, we have talked about suffix trees and suffix arrays, and we have talked first about suffixes. So, you understand how we can generate suffixes from a reference string, and we have talked about this suffix array data structure and the suffix tree data structure, and what we have seen is that suffix trees and suffix arrays enable really fast mapping because of this structure, so you can search very quickly. However, what we have seen is that a suffix tree actually takes a lot of memory and space, and in the case of the human genome, this is about 30 to 35 GB. So, if you have to use that much RAM on a normal desktop computer, you have to use a dedicated server or a computational server. Suffix arrays actually address this issue to some extent; they reduce the memory requirement but also keep the property of fast mapping because of this lexicographical sorting.

So, we have discussed this right. So, suffix trees require a lot of space memory, and this is reduced in the case of suffix arrays. What we have seen is that suffix trees and suffix arrays can map reads to multiple positions with ease. So, if you have a read that maps to multiple positions or that comes from repeat sequences in the genome, then this kind of structure is very useful for quick mapping. And this is something that will come up very often if you are looking at the human genome or other genomes like high-rate karyotes, memory, and genomes. You will see a lot of repeat sequences present. And you will get these reads from these repeat regions, and then in the mapping process, you will probably have to map them to multiple locations or identify these locations where these reads might come from. And in that case, if you are using hand stabilisation, it takes a lot of time, which will increase the time requirement substantially. In those cases, suffix trees and suffix arrays can help a lot.

So, as you have seen in the suffix tree you have these overlapping paths that are generated in the case of repeat sequences. So, you just search through one single path, and you identify all the positions where these reads can map to OK. In suffix arrays, because of the lexicographical sorting, it is again very easy. You will have multiple suffixes where you will find these read, but because of the sorting, they will be very close to each other, and you can find them very easily. So, this is the biggest strength of these methods, but what we have seen is that these methods are very poor at mapping reads that contain mismatches. These mismatches could be generated because of mutations in the read or the sample that you are working with or because of the sequencing errors that happened during the sequencing process. So, they cannot really handle this kind of read; they cannot map those reads, but of course, we have also talked about some alternatives where we can actually come up with certain methods that can allow this kind of mapping.

But given the limitation, for example, the memory and space requirement, this is something that has not been worked upon that much because what we want at the end is a method that is fast, which can handle this kind of mapping to multiple repeat positions, but also does not take too much space or memory. So, we should be able to do this mapping process on a normal desktop or laptop computer, right? That is the goal, and that is why we have not kind of gone in this direction where we are adapting this suffix tree or suffix error-based methods for mapping reads with mutations or sequencing errors. So, in the next class, we will talk about certain methods that can actually do all of these, but with very low memory



Suffix array

Search: GCG

AGCGAGCGCC#	3
AGCGCC#	7
ATAGCGAGCGCC#	1
C#	12
CC#	11
CGAGCGCC#	5
CGCC#	9
GAGCGCC#	6
GCC#	10
GCGCC#	8
GCGAGCGCC#	4
TAGCGAGCGCC#	2

TAGCGAGCGCC# 2

requirements. Thank you.