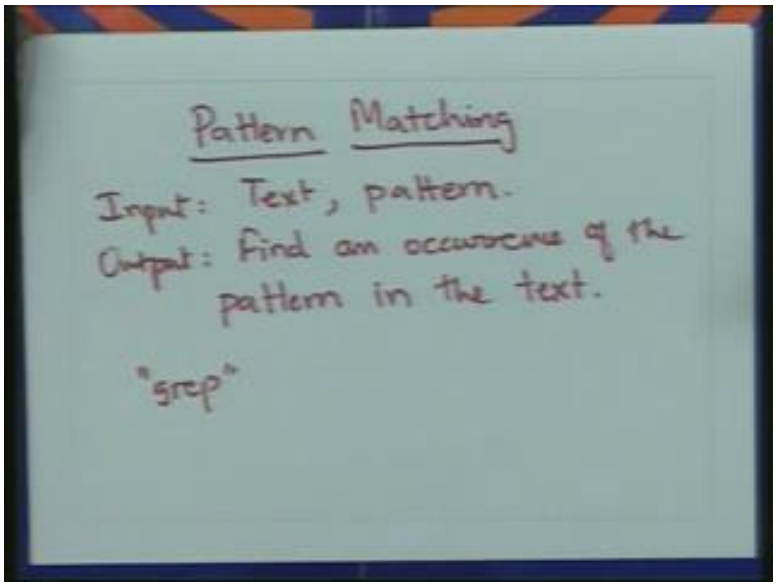
Design and Analysis of Algorithms
Prof. Sunder Vishwanathan
Department of Computer Science Engineering
Indian Institute of Technology, Bombay

Lecture - 14
Pattern Matching – I

We next look at the problem called pattern matching. This is something I am sure you used before.

(Refer Slide Time: 01:02)



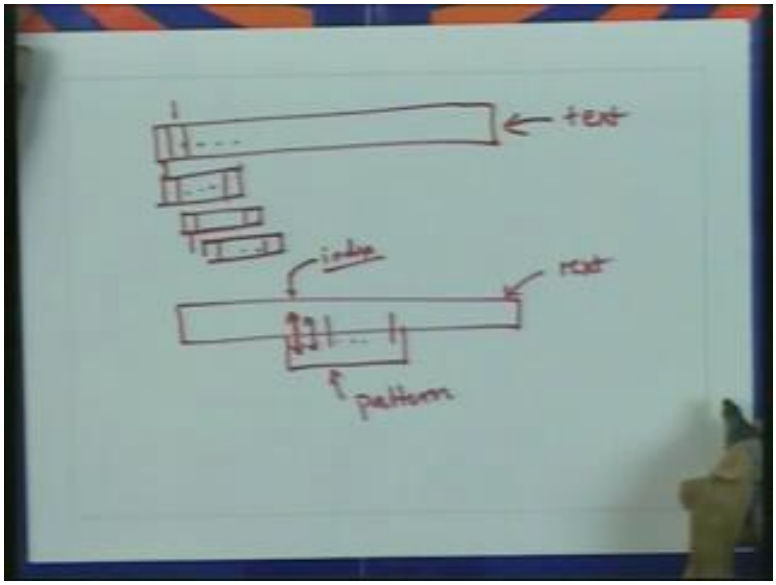
So, the problem is this you given as input, text and pattern. And you want to find, if you going to determine if this pattern occurs in the text and where. Output is find an occurrence of the pattern in the text. You could changes to say find all occurrences, it really does not matter for. So, the time being we concentrate on this, we want to find and occurrence of this pattern in the given text.

Grep is something that does it. So, I guess you are used uniques, we use uniques. Then, the command that does is it grep, you can grep for a string in a file. So, intake of file and the string as input. And look for this string in this file and I am sure every other operating system provides this facility, it is a very basic sort of facility, which is provided and which is used quite extensively.

So, what we going to do is design and algorithm for this problem, which is given a text and pattern, find an occurrence of this pattern in the given text. So, what is the first thing, first algorithm that you would write. Well, take the pattern and you sort of look for the pattern at every possible position in the text.

So, the text is say n characters long, you started the ith character in check whether, the pattern. There is an occurrence of pattern starting of the ith character do this for every i. So, for every possible position, you check whether the pattern does. In fact, occur that position you could do this in a loop. So, you can sort of scan the text, character by character starting at you know from left to right say. And for each character from this character onwards is there is a pattern exist. So, you check for this.

(Refer Slide Time: 03:53)



So, symbolically let us say this is the text, well that is the pattern I start matching character by character here. If there is a complete match, then I know that the pattern occurs here. If it does not, let us say there is a mismatch somewhere here, then I shift the pattern by one, one character to the right. And now I start comparing from this second character onwards. I check whether the second character, that text equals the first character the pattern etcetera.

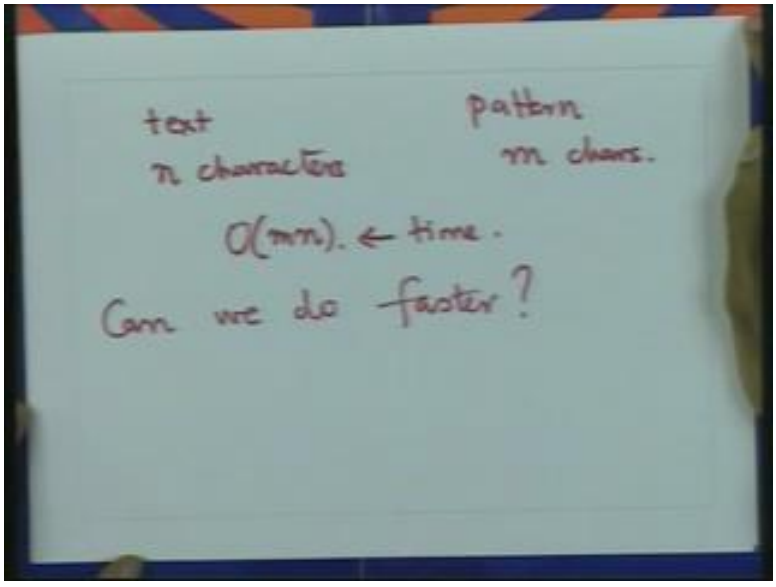
So, I check whether the patters occurs here, if it does output. If it is does not I again shift this pattern by one more and so on. So, each time I sort of shift the pattern, along the text. And well when the pattern, when the sort of pattern is at some place let us say here.

So, this is the pattern in that is the text, I just check whether the match character to character. I just check that these two character as same. If all of them are the same, then I actually found the pattern here of the text and I can output this index.

This is where the pattern occurs. So, this is sort of a ((Refer Time: 05:19)) and the sort of the first thing that you would do, first algorithm that you would come up with to search for a pattern in text. How much time I guess is the question, that we asking throughout this course and we will ask again.

So, how much time does it takes, first you this can be sort of return in a I mean I hope all of you can write code for this ((Refer Time: 0547)) algorithm is putted in the loop. And you sort of you know move the pattern across the text. So, this is should be able to do and if you doubtful I would suggest you, you just try this before we proceed further. Then, we come back to this question how much time does it take. Well, you compare let us say that of the size of the text.

(Refer Slide Time: 06:19)

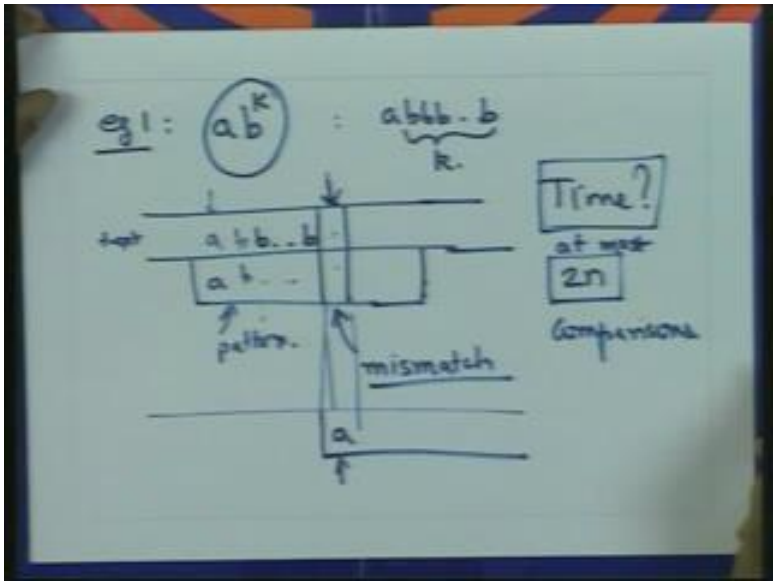


Let us say the text as n characters and the pattern as m characters, size of the pattern is m, size of the text is n. Then, for each starting point in the text we search for the pattern. So, in the loop there are I have to check for starting with n characters. And each time and the worst case I may make m comparisons. So, the total sort of time is O m n, this is the total time. This should be clear, because for each of these n characters ((Refer Time: 07:11)).

So, I when I start comparing from here I could go you know most of the pattern, I could sort of go down most of the way and then pattern discover a mismatch. In which case, you know the total time taken is o m for each of these indexes and there are o n indexes. So, well the total time is order m n or objective is to can we do faster. So, this is the question that we would like to address.

And in fact, we will do faster. So, by the time this lecture ends. You see that we can do much faster than o m n, that is you have written. To see how we can how do we go about doing this, the first thing do is to check out the few examples, which is what we will do. So, we will take some patterns, some simple pattern we will see how would once search for this pattern in a given text.

(Refer Slide Time: 08:26)



So, let us say so here sample 1. So, supposing my pattern is a b to the k, this means it is a b b b there are k occurrences of this. You can take k to be 20 for instance, real exact value of k does not matter. But, the pattern looks like this and your searching for this pattern in a text. So, let us just see what happens here. So, let us say this is the text and you started searching from this point onwards.

So, this is the pattern you sorted searching from here this is a let us say. So, there is a match, then you sort of check maybe, this is also a match. You go down you know of few characters their own matches and then there is a mismatch. So, here I have a

mismatch, before the pattern ends, which is the pattern ends here. So, somewhere here is a mismatch, which means this is not a b I know that this is not a b.

Now, what can we do well the ((Refer Time: 09:55)) algorithm you know what ((Refer Time: 09:57)) algorithm have done, it will have shifted this pattern by one character. And then, started comparing, this a would have been compared again this b. You would have immediately got a mismatch, then again go down one more you would have a again have a mismatch.

Because, well all of these are b's up to here, because the pattern is match. So, for all of these are b's. In fact so all these when I start shifting, all the initial shifts are useless in a sense. Because, when I shift by let us say this amount up to this. I know this a is go to have a mismatch with the b. So, I am going to start shifting again I am not going to find the pattern.

So, the pattern this pattern is suddenly not going to start anywhere here. If at all it is going to start here. So, if there is a mismatch I can for this pattern, I can move this all the way up to here. So, I can start comparing here now. So, I move the pattern all the way here and I start comparing. I do not need to sort of shifted one by one I can shifted all the way here and I start comparing.

So, there was a mismatch at this position I move the pattern all the way and now I start comparing. So, for this pattern this algorithm works. So, I start comparing, you know the text in the pattern, here the text in pattern. And if there is a mismatch in the ith position of the pattern, I just move the pattern all the way up to here. And I start comparing now with the first character of the pattern.

So, essentially we have some more use the factor, you know the text as match the pattern up to here. So, we know that these are all b's. And my pattern must start with an a. Since, there is a mismatch here, there is a good chance I mean there is some chance that there is an a here. So, we cannot sort of move it even further, this is what we move the text of tool.

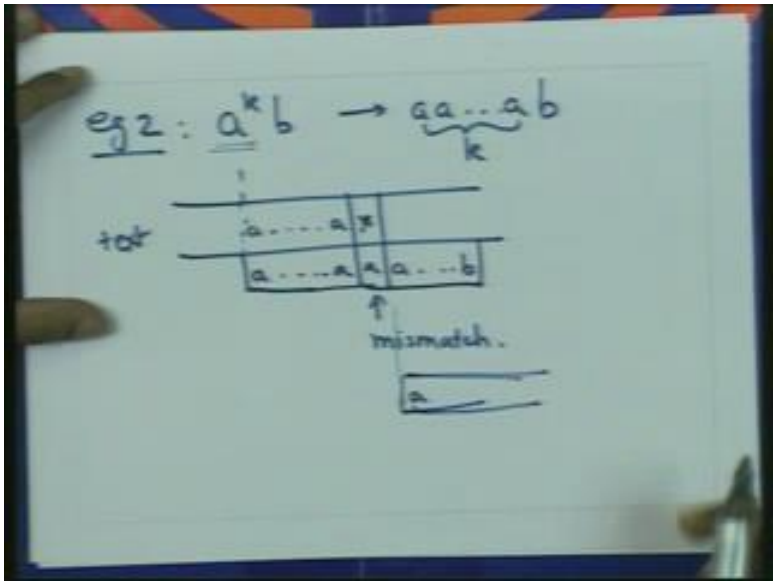
So, supposing we follow this algorithm for this pattern, if there is a mismatch I am move it all the way here. And know again start comparing, I compare this with a I go down here. Suppose there is a mismatch I move the pattern all the way here. So, how much

time does existing. Well this is the crucial argument, because some such argument will use in general.

When, the time the total number of comparisons by time will just say the total number of comparisons is actually 2 n at most 2 n comparisons now why is that. Well we look at each position of the text I claimed, that for each positional the text. I will make at most two comparisons. Why is that well, let us see we start here for these positions of the text I make only one comparison. Because, once there is a match here I never make anymore comparisons for this position in the text.

If there is a mismatch I may make one more comparison, which is the pattern shifted all the way. Now, I match a with this, if this does not match I shift the pattern and I move ahead. So, for each position in the text I make at most two comparisons. So, the time here the total number of comparisons that I make for this, that I need to make for this pattern this 2 n. So, well there is a ((Refer Time: 13:49)) that maybe we can do faster than m times n. So, let us takes another example. Again any simple example, but this will again illustrate what we are trying to do.

(Refer Slide Time: 14:05)



So, example 2, so earlier we took a b to the k that is now take a to the k times a to the k b which is nothing but a, a, a k times followed by b. This is the pattern we are looking for in a text. So, how much time does it take, how many comparisons do we have to make, if we sort of do things not just namely, but in an intelligent way.

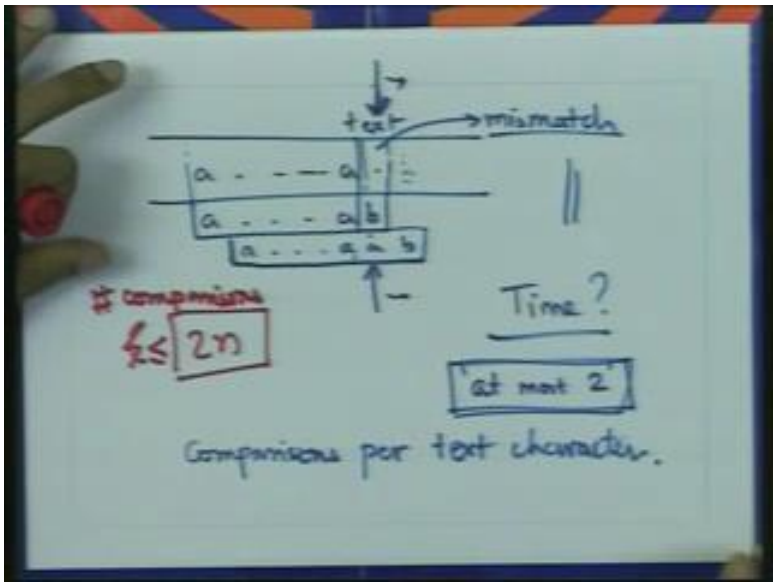
So, again let us see what happens. So, here is my text I start comparing, let us say from here. I have done all the way up to here and I am sort of comparing from here. And let us say that is the pattern and I have a mismatch here. So, I have a, a, a here all the way up to b this is the pattern. In the text I know that I have a is here. Here, there is a mismatch, this could be anything. This is the mismatch well what can I do.

So, in the ((Refer Time: 15:28)) sort of thing I would have shifted this by one, this would have shifted by one and I will was started comparing again. And you know that again there will be a mismatch here. There will be matches all over here and there will be a mismatch here. This is known, because I know what the patterns. So, if you want to do it intelligently, you know that starting the pattern anywhere here all the way up to here is useless.

Because, I need you know string of size a is of size there should be k a's and they do not occur here. This is not an a, so I can in fact, start the pattern right here. So, once as a mismatch I can start comparing right here. So, I can move the pattern all the way after the mismatch. Once, as a mismatch I just move the pattern up to here. So, I know because the reason is I cannot find pattern, anywhere here I have to start here.

Because, pattern has to start with an a and then I need to see k a's. And I do not see that here, this is not an a. So, again I use some information about the pattern to move the string all the way. There is one more case here, which is when the mismatch is at b. But, before we do that let us maybe do that and then come back and see how much time takes.

(Refer Slide Time: 16:54)



So, here is my text now the mismatch is here. So, this is the mismatch, but text matches all the way up to here, except you know there is a mismatch now what do I do. So, now there is a good possibility, that this is a. There is a good possibility that the place, where there is a mismatch, in the text it actually is a. In which case, if this is a b the next thing is a b, then you will miss I mean you cannot move it by more than one.

So, I have to try this the pattern shifted by one. This is quite possible a, a b if there is a mismatch at this position, it is quite possible that the pattern occurs here. So, I can at most shift the pattern by one. So, this looks like you know worst case I cannot shift by more, the pattern by more. This looks like a worst case and be a back to looking something looking at the ((Refer Time: 18:12)) algorithm.

But, while we are shifting the pattern by one, there is something you are gaining. What we are gaining is that, we know that there are up to this position there is a match. While I shift the pattern by one I do not have to again match all of these as I did in the ((Refer Time: 18:31)) case. Remember, then ((Refer Time: 18:35)) algorithm I shifted by one, then I started my comparisons here do this match, do this match, do this match and so on.

Here, I do not have to do that, I can start comparing from here. I shift the pattern by one and then I start comparisons here. So, this is an a if there is a match, then I move up again, this fellow moves up a again this moves. Now, I see whether this is b, if there is a

mismatch I am moving the pattern by one, but the text pointer remains where it was. So, the text pointer remains where it was.

So, how many, so I hope if the algorithms in this case is clear it is slightly more complicated than a b to the k. But, I hope the algorithm is clear. So, how much time does it take. So, what is the time, which means how many comparisons do we make in the case of a to the ((Refer Time: 19:35)).

So, let me again let us over this ((Refer Time: 19:37)). Suppose, this a mismatch in one of these positions I move the pattern all the way here. I the move the pattern here, if there is a mismatch at b well I am move the pattern by one by one unit. But, I start comparisons here, this takes with this pattern. Now, of course there could be a mismatch, if there is an mismatch, then I move the pattern all the way up.

If there is a match, then I move this point by one, this point by one and I compare. That is the ((Refer Time: 20:10)) algorithm works. How much time does it takes, well the claim is again for look at each position in the text. By look at this position in the text, then the number of times is compared is at most 2. If there is a match, then the text pointer moves forward. If there is a match text pointer moves forward and that position is never compared again.

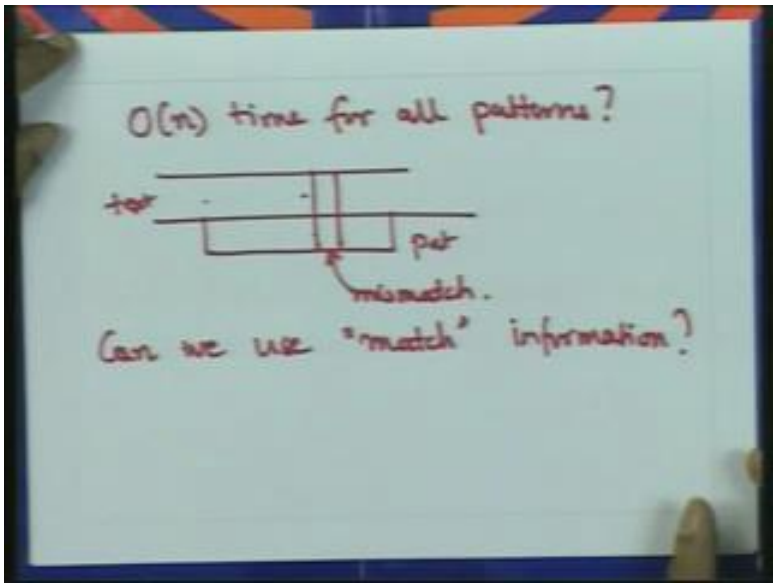
So, it at most once a match occurred at a position with text at most once. How many times can a mismatch occur, will again I claimed that the mismatch will occur at most once. Well, if the mismatch occurs at a position, which is not b at one of the a's, then the text moves forward. So, this is the case, if the mismatch occurs at one of the a's, then the pattern is moved all the way up here.

And the text pointer, which was here now moves up by 1. If there is a mismatch text pointer moves by 1 I never compare this again, I never compare this character again. If there is a mismatch at b, the text is again here, but the pattern has moved. Now, check what can happen, either there is a match in which case the text moves or there is a mismatch at an a again the text moves.

So, there can be at most two comparisons per text character. So, for each character in the text, there can be at most two comparisons. So, the total time is 2 n number of comparisons. So, number of comparisons at most 2 n. So, in both cases we see, that the

number of comparisons is at most 2 n. And this gives as scope, that maybe we can do it you know faster. For instance, it is ((Refer Time: 22:47)) asking can we do it o n time for all patterns.

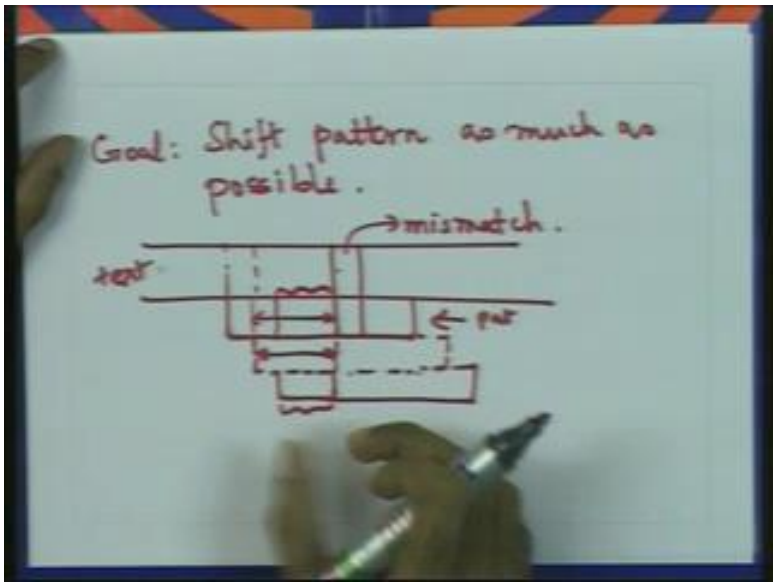
(Refer Slide Time: 23:00)



So, what is the scene, suppose it should be able to do this, what is the scene? Well, the question is... So, if I have this is the text and I have, now this is the pattern. And I match some of them and here I have a mismatch. In each of the earlier cases, we some more used this matched information. This information, that this portion of the string matches with this portion the text was used in both cases.

So, this is what we would like to use. So, can we use the match information, which means the information that this portion of the string actually match this portion of the text can we use this. And this is what we would like to do, somehow use this information. What you would also like to do, like in the previous case is try and shift the pattern as much as possible.

(Refer Slide Time: 24:23)



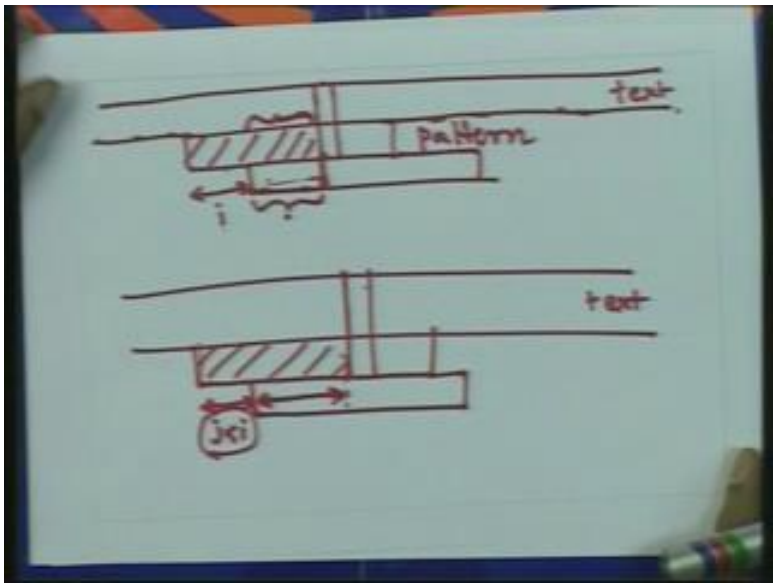
So, here is one goal, so shift pattern as much as possible. So, what does this mean, I mean. So, here is my text we going to see many of these two line highways throughout this, this stock. So, here is my pattern in fact, if you see this figure this is a text in ((Refer Time: 24:54)) pattern I am not going to keep writing this text pattern will miss. Now, let us say this is a mismatch here.

Now, when can I say that moving it by one is not necessary. When can I say that this is useless. Well, this is useless if this portion will not match this portion of the text. If this portion does not match this same portion of the text, there is no point and moving it by one. We in some ways no this portion with text, because it has exactly match this portion of the pattern.

All this I have a match, in the text up to here I have a match this is my first mismatch, which means if this portion of the pattern, does not match with this portion of the pattern. Forget the text for the time being, just look at the pattern. This is the original pattern and here is the pattern shifted by one character. If this portion of the pattern, does not match this portion of the pattern, there is no point and shifting it by one, because if there is a mismatch somewhere here, there is going to be a mismatch there also. How about two characters, again it is a same thing only now I am concerned about this portion of the text, this portion and you see that portion of the pattern. In general, so let me draw this again, well the crucial idea again is that. Once I have matched up to here, then whether to

shifted by one or two or three or four it is only something do with the pattern, it has noting do with the text in some sense, because the text is already matched. So, this decision I can make by looking at only the pattern what do I mean by this.

(Refer Slide Time: 27:20)



So, here is my pattern. So, thing of the text sitting up there, I have a mismatch here. And let us say, you know the best way to do it is to shift by you know some i units. Essentially, I do not by shifting I do not want to miss and occurrence of the pattern, in the text. So, that is what I do not want to miss, otherwise I would like to shift as much as possible.

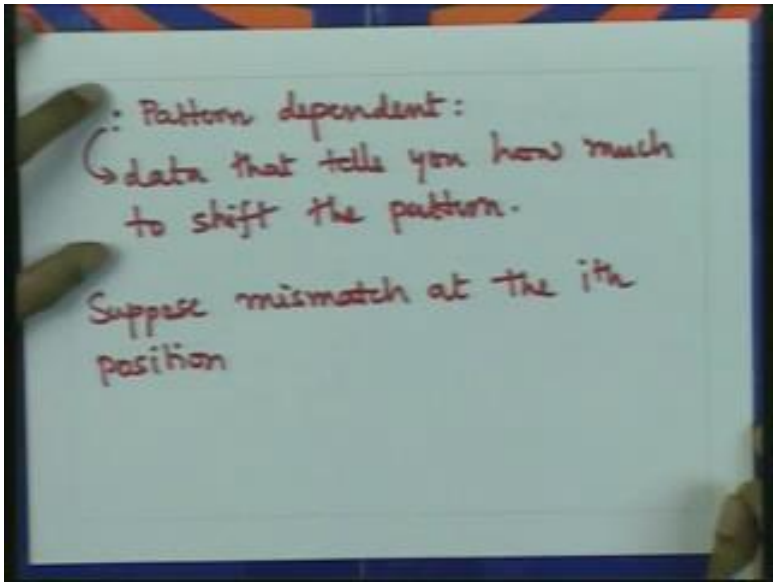
So, here is the text, supposing this is shifted by i units here, supposing you know if i is the best you can do, what is this mean? This means, that this portion must match this portion. This portion which is a prefix of the pattern, must match the suffix of this portion of the pattern. This prefix must match this suffix and this must be the largest prefix I cannot have a larger prefix matching this, for instance if instead if I move the pattern by less. Let us say I minus 1, then I have a and there is a match, then I should not you know miss that opportunity. So, what I am saying is this. So, here is my text, so here is the pattern, let us say this is shifted by j, which is less than i, this is j which is less than i. The mismatch is at the same position, this is the same position is that.

Now, I can safely do away with this j's, if this is not a prefix. This prefix is not a suffix of this. So, for all j less than i this prefix is not a suffix of this portion of the pattern. So, this

is this portion of the pattern is the same as this, these two are the same. So, any prefix which as larger than this length, larger than this will not match suffix, then I can move it forward.

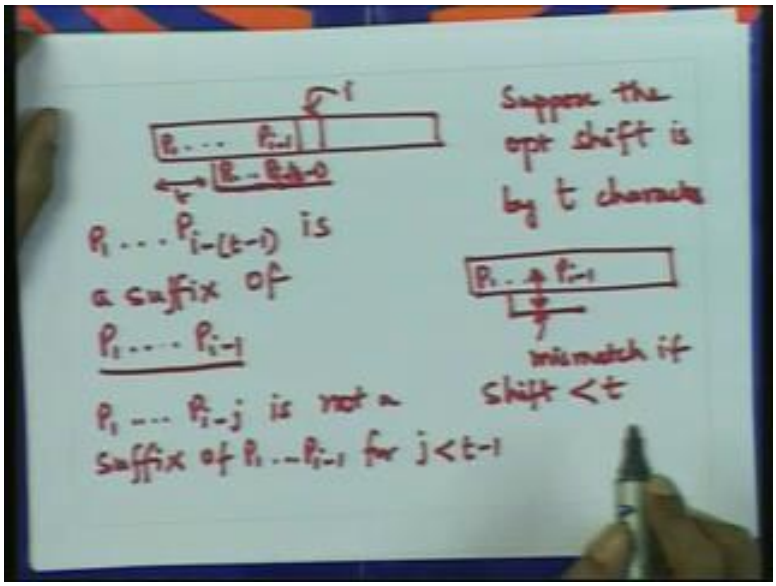
And essentially I want you know the largest prefix of the pattern, which matches the suffix at this point. So, if there is a mismatch that is what I want. Then, I can move it all the way up to that portion and then start comparing. So, this is what I really want to do. So, this data is only pattern dependent, that is the first thing.

(Refer Slide Time: 31:32)



So, this only pattern dependent this is the data, that tells you how much to shift. And what is this data. So, supposing you have a mismatch at the ith position. How much to shift the pattern by that is what we want to know. So, how much to shift the pattern by well.

(Refer Slide Time: 31:45)



So, let us look so here is the pattern, this is the ith position I have p 1 through p i minus 1. And supposing I have an optimal shift, which looks like this. So, p 1 so on and this is p i minus, now let us say t minus 1. So, supposing the optimum shift is by t characters, which means if I shift by less than something less than t character. Then, this is not going to work I mean in the sense there will be a mismatch.

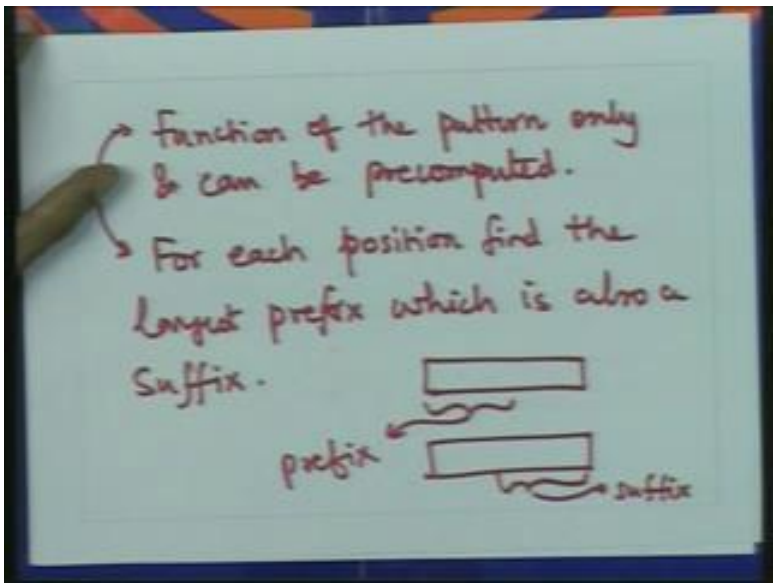
So, what is that mean. So, if it is less than t to here is my pattern p 1 to p i minus 1. If I shift with by less than t then there will be a mismatch somewhere here, mismatch if shift is less than t. So, this means that there is no point and time out the shift, I am it as well shifted all the way up to t.

And if the optimum shift is t, this means that you know there is I have a match all the up to here, which means p 1 and so on up to p i minus t minus 1 is a suffix of p 1 to pi minus 1. So, this is the word and that this is the word with i minus 1 characters. And this word is a suffix of this, that is what this shows that these two of the same, these two of the same and so on, and because it is a mismatch somewhere here. We also know that p 1 through p i minus, let us say j is not a suffix of p 1 through p i minus 1 for small for the values is j which are smaller than t minus 1. So, for j less than t minus 1. So, the values of j is less than t minus 1. So, the shift is less than t which means, the shift is less than t. Then, this is not a suffix, when I shifted by t units, then I have a match all the way, which means this prefix of this pattern is also a suffix of p 1 through p i minus 1.

So, given a pattern what we trying to do is for a mismatch. We want to find out how much we can shift the pattern by, we would like to shifted by as much as possible, without missing and occurrence of this pattern in the text, that is what we are doing. So, supposing the... So, let us look at this figure. So, supposing the mismatch is at the ith position in the pattern, you the text sort is to here there is mismatch at this position.

How much should I shift the pattern by. And we saw that, I would shifted by t units, this is t units. If the following thing holds, which is this prefix p 1 through p i minus t minus 1 i minus t plus 1. This prefix is a suffix of this portion and this is the largest such string. If I shifted by less than this, then there will be a position where there is a mismatch, that is what it says. For all smallest shift there will be a mismatch, here I get a perfect match, there is a match between these. So, just says that this is the longest prefix of the string, which is also a suffix of the string. I want actually then to be proper prefix of this, which is the suffix. So, then the crucial thing, the thing that we want to find out this is.

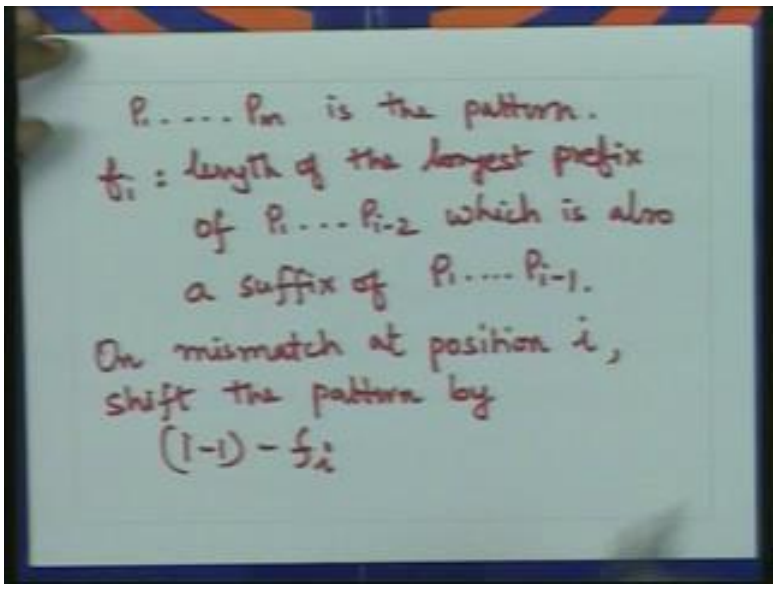
(Refer Slide Time: 37:02)



So, the first thing is this quantity is a function of the pattern and can be pre-computed. So, I can pre-compute this value as to for each mismatch how much to shifted by. So, what is this function of the pattern, that I want to compute is for each i, for each position let us say find the largest prefix which is also a suffix. So, I have actually not defined prefix and suffix I hope you know what it is?

So, given a string a prefix is all the initial portions, the initial the first i portion is a prefix. And suffix is just a last portion, this is suffix and this is the prefix. So, given any string this is the prefix and that is the suffix. So, given any string I want to find the largest prefix, let us say largest proper prefix which is also a suffix. Clearly, if I take a string, the string is a prefix of itself and suffix of itself. But, I want the largest proper prefix of a string, which is also a suffix. So, this is what I want to compute. Supposing, I have computed this.

(Refer Slide Time: 39:05)

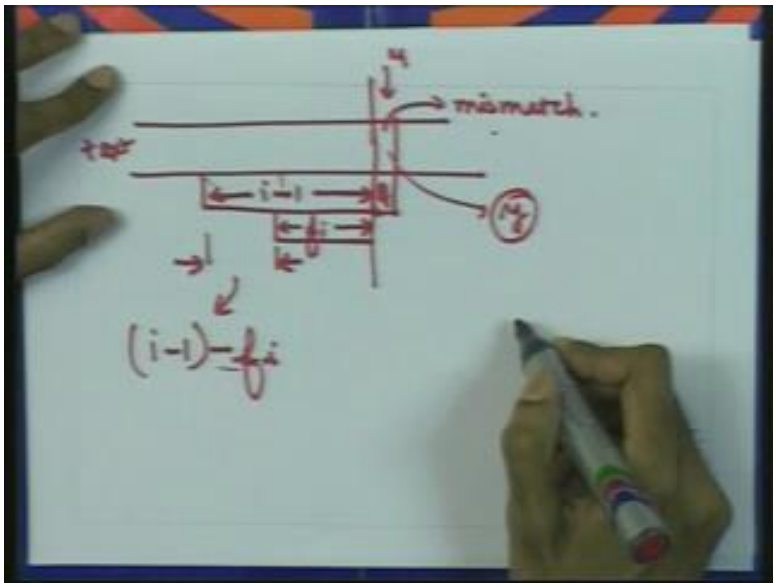


So, I have pattern p 1 through p n is the pattern. Let us say f i is length of the longest prefix of p 1 up to p i minus 2, which is also a suffix of p 1 up to p i minus 1. This p i minus 2 is just to make it proper prefix. So, any prefix of this. So, length of the longest prefix of this, which is also suffix of this I could have written p i minus 1. But, then I will mention the longest proper prefix.

So, supposing I have computed this for each i, for each position in the pattern of computer this f i, which is the length of the longest prefix of p 1 through p i minus 2, which is also a suffix of this, which means. If I take any other prefix which is longer, than f i then it will not be a suffix of this pattern. Now, on mismatch at position i what can I say?

I say, that you can shift the pattern by i minus 1 minus f i. I can shifted by this much, given f i which is defined this way. Then, on mismatch at position i, I can shift the pattern by i minus 1 minus f i. So, let us see this.

(Refer Slide Time: 41:31)



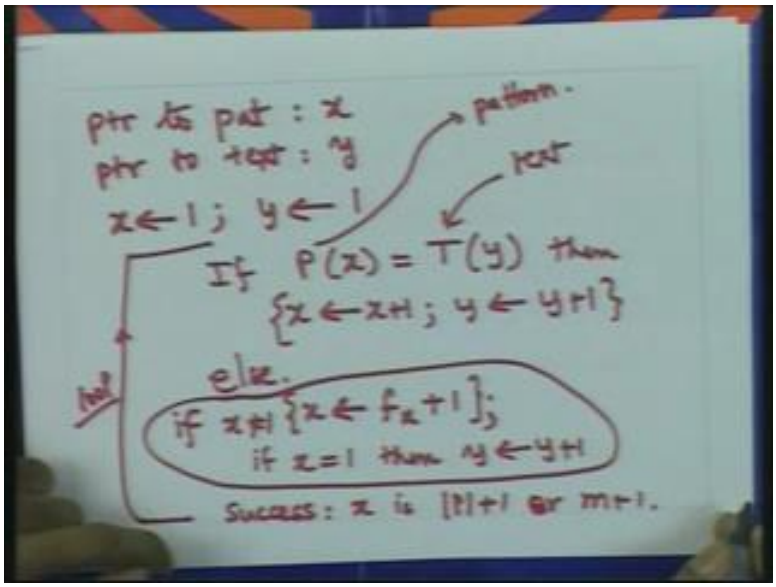
So, here is my text, this is the pattern and this is the ith position in the pattern and there is a mismatch. Now I know, so let us see what this is ((Refer Time: 41:58)) this says f i is the length of the longest prefix, which is also suffix. So, let me if I put the term here. So, this is the string that I am considering now. I am only looking at ((Refer Time: 42:15)) p 1 through p i minus 1 the mismatch is at i. But I am looking at p 1 through p i minus 1 good.

So, this is the string I am looking at. And I know that which is the longest prefix of this same string, which is also a suffix. This I know as length l i, which means if I move the pattern all the way up to this position. Then, there is a match here. If there is a match here I know that the text as match everywhere here. So, there is a match in the text I also know, that this is a longest prefix which means if I shifted by any less, there will be a mismatch at some position, which means you have be a mismatch in the text.

So, the among time move is this the among the shift the pattern is this, this is i minus 1. So, among the shift the pattern is i minus 1 minus l i which is what we had f i this is f i. So, I must shift this by i minus 1 minus f i. So, this is what I have said in the previous step. So, once I compute these f i's if there is a mismatch of the ith position of the

pattern, if p i mismatches the text. Then, I just shift the pattern by i minus 1 minus f i. I start comparing now. So, let us right this code ((Refer Time: 44:02)) see what this terms out. So, initially I am looking for a pattern in a text.

(Refer Slide Time: 44:12)



So, the pointer to the pattern I will refer by x, the pointer to text I refer by y, the variable will be y. Initially my initialization is x is set to 1, y set to 1 and a start comparing. Now, what is the generic step look like. So, if p of x is text at y, this is a text, this is the pattern. If there is a match then what you do? Well, I need to increment x and y by 1 and then continue. Then, x is set to x plus 1, y is set to y plus 1 and I continue, I will have to go back in a loop.

What happens otherwise, if there is a mismatch, then I know that I am a shift the pattern by this much ((Refer Time: 43:35)) I shift the pattern by i minus 1 minus f i. So, this is where the mismatch is... So, this is where my y this pointer on the string is y, the pointer in the text was i here. Now, what do I set it to, I set it to f i plus 1, if it your I initially I set it to f i plus 1. If it your x initially I set it to f i plus 1.

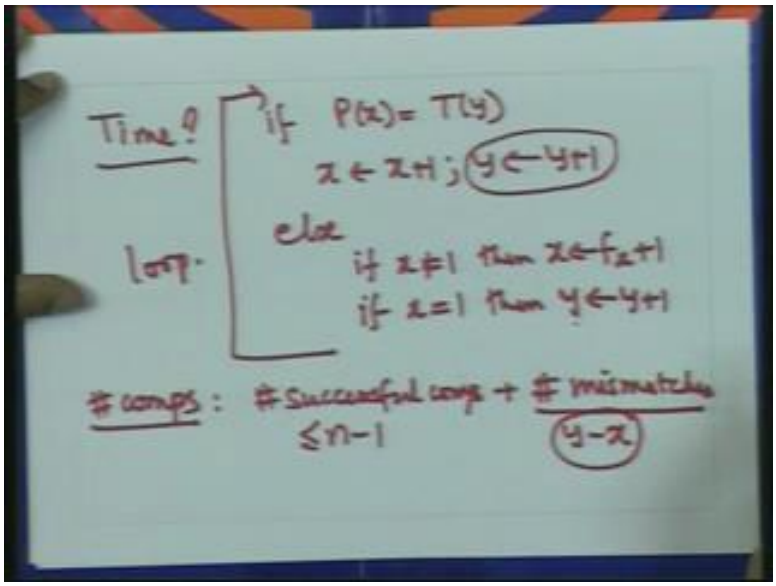
Because, I have moved it by i minus 1 minus f i or i minus 1 minus f x remember my point of ((Refer Time: 46:21)) x here it does not i. But, x if there is a mismatch, I shifted by x minus 1 minus f x. So, the point next character I am going to check in the pattern is nothing but f x plus 1. So, let us do this. So, else x is set to f x plus 1, this is what I want to do.

There just one case that we need to take care of which is, this actually is true when x is not equal to 1. So, this is when x else, if x note equal to 1 then you do this. Now, if x equals 1, what is this mean? Let me the first character of the pattern, there is a mismatch with the text. And comparing the pattern with the text, if x equals to 1 is the first character.

So, the first character of the match pattern mismatches with the text. Then, I just move both pointers I move the pattern by one and I move the text pointer by one. So, let me right this all it is inside else. So, if x equals 1 then y is y plus 1. So, I have I just increment the text pointer by one. So, all this it is inside the else. So, this is inside the else and the whole thing is in a loop.

When do have success, when do I stop. Well, I stop when x becomes size of p plus 1. So, let me write that to success, if x is size of p plus 1 or m plus 1, which means I have match successfully the m positions of the pattern. Then, I have found the pattern in the text, this is the success and the whole thing is in a loop. So, in a while loop you can check whether x is p plus 1 or not. So, this is the procedure this is the algorithm given f x. In this function f x for each f i for each position i. Then, the algorithm for searching, this pattern in the text is this. And now let us see how much time this takes?

(Refer Slide Time: 49:31)



So, what is the time, the time let me quickly write the main loop again, it is says if p x equals T y, then x is made x plus 1, y is made y plus 1 else the crucial step is... If x not

equal to 1, then x is f x plus 1. So, you shifted the pattern if x equal to 1. On the other hand, then y is y plus 1 in the new loop, this is algorithm and then, you check for success.

So, how much timed does it take, well it is a number of times these statement are executed. So, how many times are these executed? Well, I can count number of comparisons, this will tell me the order of the algorithm. Now, the number of comparisons, that I make totally over is nothing but number of successful comparisons, which is number of matches plus number of mismatches.

So, now let us look at each of these in term, this is very similar to the merge sort. So, I would, once you look at this analysis go back and check out merge sort. So, now what is the number of successful comparisons. Now, each time there is a successful comparison, why increases by one, check out that y increase by 1 y never goes down. The pointer on the texts never goes back always goes forward.

So, the number of comparisons each time y goes up by one it is start set 1 ends up at n. So, this is at most n minus 1, because y this smallest value is 1, the largest value is n. And each time there is a successful comparison it increases by 1. How number of mismatches, well in one case x drops by at least 1, in the other case y increases by 1.

So, if I look at y minus x, if I look at this function y minus x. This always increases, when there is a mismatch. If there is a match y minus x remains the same both of them go up by 1 nothing happens to this. If there is a mismatch, this increases initially this is 0, the largest this can be is is at most n minus 1. So, number of mismatches is also n minus 1.

The other way to see that the number of mismatches at most n minus 1 is that each time there is a mismatch. I shift the pattern by at least one. Remember, there is a mismatch I shift the pattern. So, I shift the pattern by at least one, the number of time I can shift the pattern is at most n minus 1. So, the number of mismatches, number of unsuccessful comparisons is at most n minus 1.

So, the total number of comparisons is twice and minus 1, which is what we had for those small examples. But, this now shows that true for any example. All we need to do

now which we do next time is to compute this function f x. Once we compute that we are through.