Data Structures and Algorithms

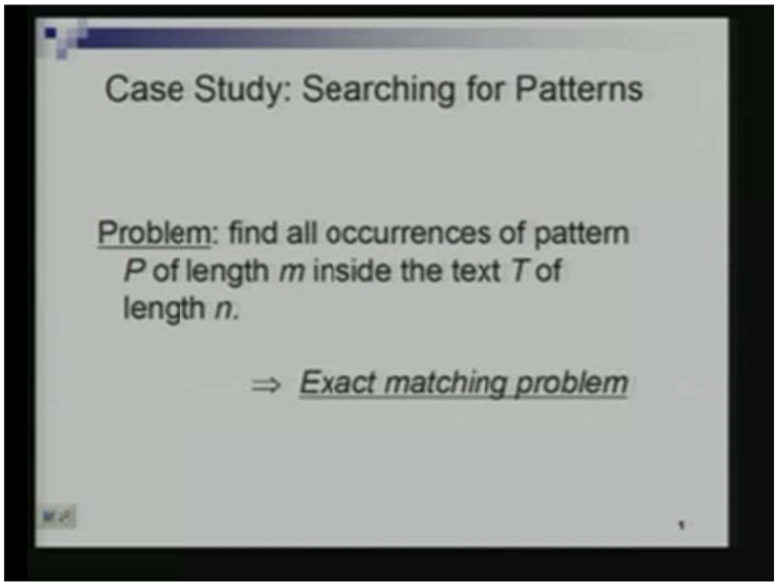
Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 17

Case Study: Searching for Patterns

So today we are going to be talking about pattern searching or pattern matching. The setting is that we are given a piece of text. Let's say 't' is our text which is of length 'n'. So it is 'n' characters over some alphabets. So for today let's assume that it's the English alphabet. We have to search for a pattern 'p' in this text. You have all come across this problem. You are using a certain editor. You want to find out all occurrences of a certain work. Most editors will provide you the facility.

(Refer Slide Time 01:17)


Case Study: Searching for Patterns

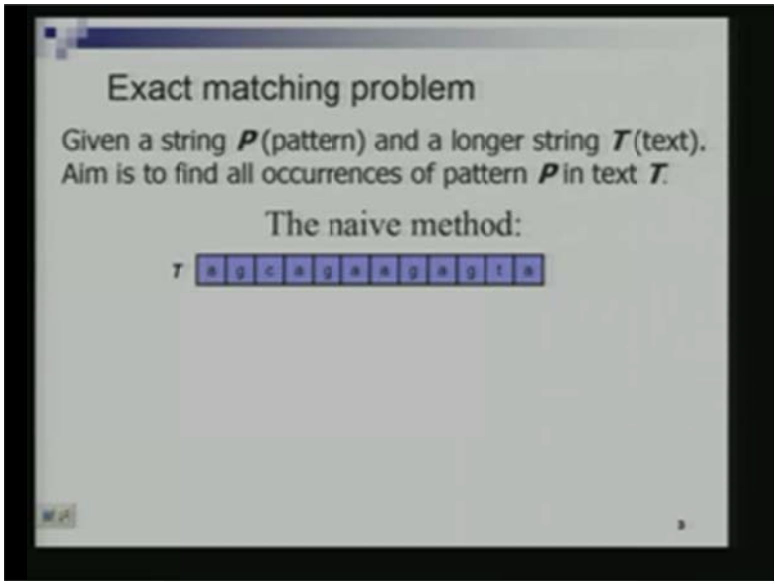
Problem: find all occurrences of pattern *P* of length *m* inside the text *T* of length *n*.

⇒ *Exact matching problem*

The browser provides you the facility. Many things provide this kind of facility that you can search for a particular pattern. Not one occurrence but all occurrences of the pattern. So question is: how are we going to do it? This is also the exact matching problem. There are versions where you cannot do an exact matching. As in you can search for the pattern in an approximate manner. That means occurrences of the pattern in that text where it matches most of the texts. If there is difference in one character, it is okay. Difference in two characters, it is okay. But at most other places it matches.

So how would one solve this problem? What are the applications? This comes up in text editing. As you can imagine, it comes in information retrieval. One big application is bio informatics where your text would be a large database with sequence of nucleotides in it and you are searching for a particular gene or a particular sequence of nucleotides and you want to find out where all they occur. So this is an example where the alphabet that you have is small. If it's a DNA then the alphabet has 4 bases in it. But if it is a protein sequence, then it has 20 bases or it is an alphabet of size 20. So what is the naïve method of doing this? Suppose this is the text and I have a certain pattern let’s say the pattern is ag ag. This is the text.

(Refer Slide Time: 03:29)



Exact matching problem

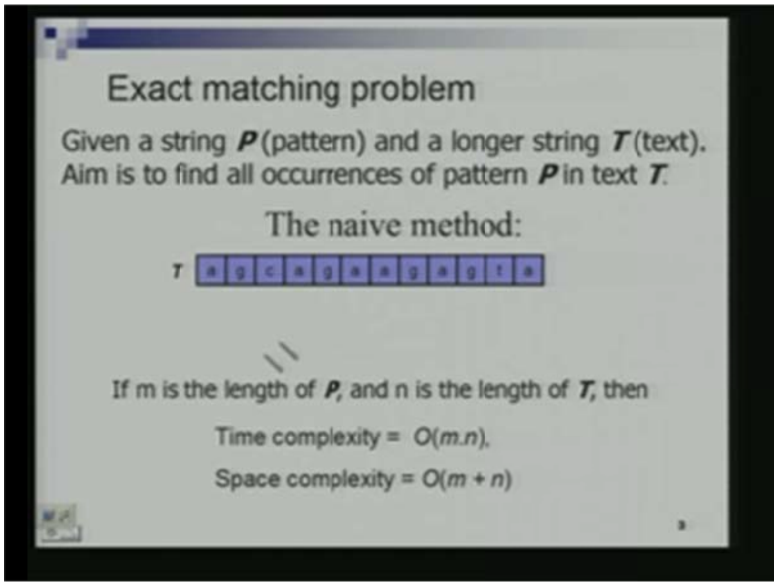
Given a string ***P*** (pattern) and a longer string ***T*** (text).
Aim is to find all occurrences of pattern ***P*** in text ***T***.

The naive method:

So what do you mean by the naive method? I will put the pattern at the first place and I will start matching the pattern against the text. In the third place, I find the mismatch. So what do I do? I start again from the second element. Is it clear to everyone why I should start from the second position and not from the fifth position? What we are going to do is move the pattern by one step. We will see examples of why this one step was critical and not more. Clearly by moving it by one step, we are not losing anything. We will be able to find all the occurrences of the particular pattern. So we do this again. There is mismatch right at the very first step. So we will move forward by one step and now here you see this is the mismatch of the fourth position. So how much should I move it? We will keep moving by one. Even here there is a mismatch at this place and so you will say that this is one occurrence.

So this is the question. Why are we moving by 1 and not by 4? If we were moving by 4, we could have skipped this thing. So suppose I moved at 4 at this position, then where would I end up in? It is a g a g here and then if I moved at 4 again, then I would end up a g a g and so I would have missed the occurrence of the pattern because the pattern is not occurring at multiples of 4 or such things. It could occur anywhere in the text. So moving by 4 is a wrong thing to do.

(Refer Slide Time: 06:40)



Exact matching problem

Given a string ***P*** (pattern) and a longer string ***T*** (text).
Aim is to find all occurrences of pattern ***P*** in text ***T***.

The naive method:

T
a g c a g a a g a g t a

If m is the length of ***P***, and n is the length of ***T***, then

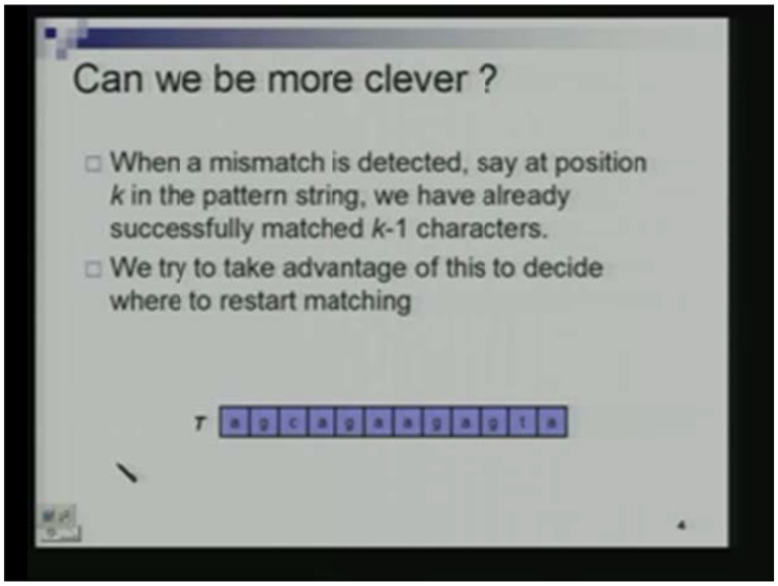
Time complexity = $O(m.n)$,

Space complexity = $O(m + n)$

How much time does it take? n m. So for every position that I put the pattern, I might have to match up to n places. It might be that only at the very last place, we find out the mismatch in which case I am matching it at all the places of the pattern. How long was the pattern? So for each location of the pattern we might have to do ‘m’ matches and how many different positions of the patterns are there? n different positions. So I might be spending as much as m n time. The time complexity then is order m n. space complexity is also going to be an issue. How much space we require? Is this clear that we require only m. we just need to store the text. We need to store the pattern. We don’t need to any other additional space. We don’t need another array to move the pattern in it.

This moving the pattern is only to show it to you here that you can always increment your index appropriately. So this unfortunately is not too good. Why is it not good? It’s too high. So if you have a huge document, m is very large and your pattern is also reasonably sized. Let’s say 20 or 30 characters, then you are going to spend a lot of time. So this is not the way it is done. We are going to look at another solution today. We are going to look at a third solution in the next class. So this will be the sequence of improvements. We will improve the time complexity today from m n to something else. I will see what that is. So how can we do better?

(Refer Slide Time: 07:48)



Can we be more clever ?

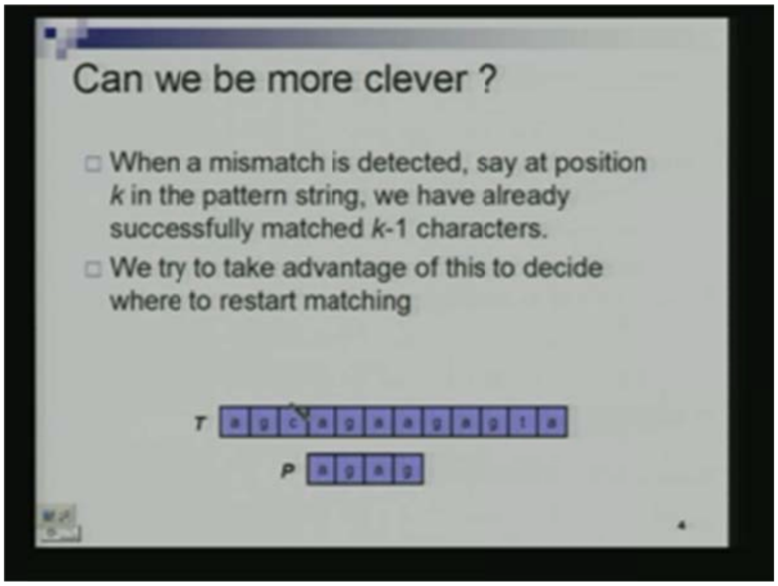
- When a mismatch is detected, say at position *k* in the pattern string, we have already successfully matched *k*-1 characters.
- We try to take advantage of this to decide where to restart matching

T	a	g	c	a	g	a	a	g	a	g	t	a

So when a mismatch is detected at a certain position, let’s say at position ‘k’ in the pattern string, what do we know? We know that we have matched k-1 characters before that and I will try and take advantage of this fact that we have already matched k-1 characters. Let’s see what I mean by this. So here the mismatch is detected at position 3 of the pattern. So what do I know? I know that the first two characters of the pattern are the same as the last two characters of the text. So I can use this information to try and move the pattern not by one step but by more steps. Why? Let’s look at the second. So this is c. There is a mismatch here. So what do I know? Even if I did not know what was here, I know since this is the mismatch I know that here, at this position in text it is a ‘g’ and this position text it is an ‘a’.

Now if I shift the pattern by one unit what happens? I know that in this position in the text it is a ‘g’. You understand why I know that? I don’t have to look at the text. Now that although the text is here in front of you, you can erase these two things and yet you know that this is ‘g’ and this is ‘a’ and since this is a ‘g’, there is no pointer in shifting in one step because then this would never match with this. We will have to shift by one step if this were also ‘a. If this were in ‘a’, then we would have shifted it at by one step. So that’s the idea and we will see how to implement this idea. Now again what do we see? We see that the mismatch at the very first step. The mismatch is at the fourth position of the pattern. Now should I shift it by one position? Should I shift it by two positions or three positions?

(Refer Slide Time: 11:41)



Can we be more clever ?

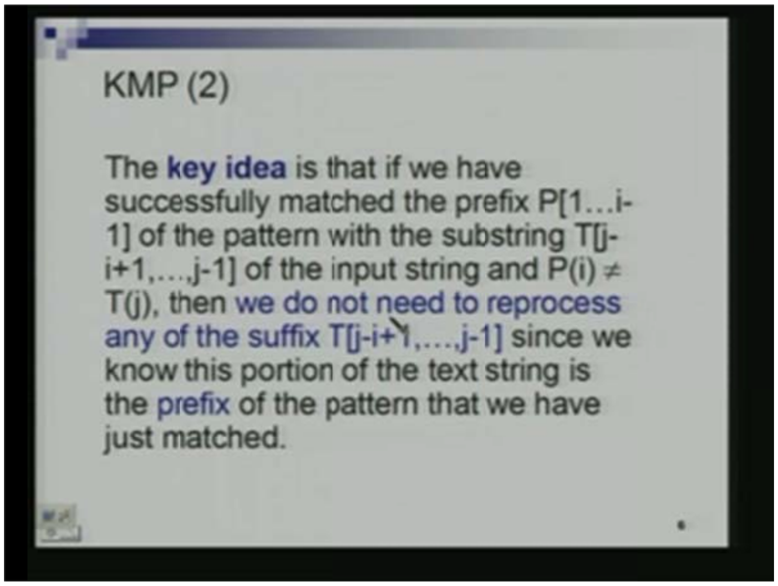
- When a mismatch is detected, say at position *k* in the pattern string, we have already successfully matched *k*-1 characters.
- We try to take advantage of this to decide where to restart matching

Now what do I know? So what I claim is you don’t have to look at the pattern here. You know that the last three characters of the pattern are the same as these three characters here because after all you know the mismatch was happening at this position. So this has to be an ‘a’. This has to be a ‘g’ and this has to be an ‘a’. Now if I were to shift it by one position, then that means that I have to move by one position. So there would be a mismatch. If I were to shift it by two positions, then I know this is an ‘a’ this is an ‘a’ and what else do I know? I know this will become a g. but the g was already mismatching with this. Actually I don’t really know that. The shift will depend upon the pattern of course and we have to determine what the shift will be. So here it’s completely clear that we would not want to shift it by one unit. But whether we wanted to shift it by two units or not, we don’t know. I am shifting it by two units. We will see that exact procedure very shortly. That’s what the purpose of this lecture is.

Once again we see that there is a match here and then there is a mismatch here. So mismatch is at the second position. Should I shift it by one position? I will shift it by one position again. Now I see a complete match. There is nothing to be done here but I have to continue. I can’t stop. How much should I shift it forward by? 2 or 4? What do you mean by split pattern? What do you mean by mix pattern? If there was another ag here, this are two occurrences there.

This would be one occurrence and this would be another occurrence. If this were a and g, it’s all occurrence of this pattern. I am not saying all disjoint occurrence of the pattern. I should be shifting by two units. So this is what is known as The Knuth-Morris-Pratt Algorithm. It’s very famous algorithm so key idea is when mismatch occurs we need not restart the computation all the way from the back. We can use some information about the pattern. Some information obtained from the pattern to determine how many steps we should shift forward. What we are going to do is we are going to construct an array ‘h’. That’s going to determine how many characters to shift the pattern to the right in case there is a mismatch. So we are going to store this information of how many characters we have to shift right in an array which we will call ‘h’ and we will see what the array ‘h’ would be.

(Refer slide time 13:50)



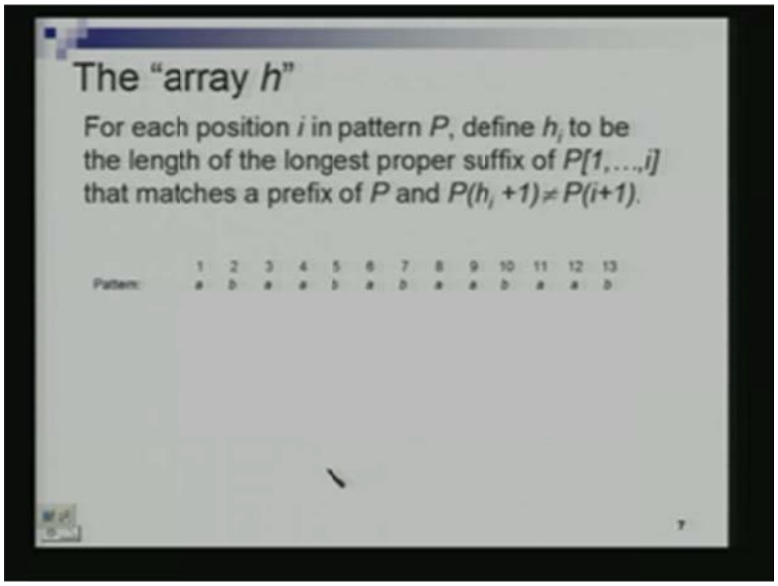
KMP (2)

The **key idea** is that if we have successfully matched the prefix P[1…i-1] of the pattern with the substring T[j-i+1,…,j-1] of the input string and P(i) ≠ T(j), then we do not need to reprocess any of the suffix T[j-i+1,…,j-1] since we know this portion of the text string is the prefix of the pattern that we have just matched.

What is the key idea now? So I guess I need to explain it more clearly. So recall we have our text. We have a pattern. We have seen mismatch at a certain position. That means this is not the same as this. Suppose if we have successfully matched the prefix P1 through P (i) -1 of the pattern with the substring T (j- i +1…… j -1), so let’s label these things. So this would be location j. This would be i. So I successfully match P1 through i-1. So this would be location i-1. I successfully match P1 through P (i) - 1 of the pattern. With what part of the text? It’s j-1. This would be j-1 and basically we are counting i-1 locations here. So that would be location j-i+1 to j-1. So this is the part of the text which is the same as this pattern. That I know because this is the first place where the mismatch happens and pi pattern this is not equal to tj, then we did not reprocess any of the suffix.

So what we are saying now is that we need to determine how much we need to shift. So we should shift in such a manner so that this part matches with this. So now this is the shifted pattern. I need to shift in such a manner so that these two parts are the same. This part matches up completely. Why would this part match completely? This is because then this would also match up with this here because I know this is the same as this. I will repeat it with the next slide. So with each position of the pattern, we are going to define. So recall we said that there is an array ‘h’ which determines how much we are going to shift the pattern right.

(Refer Slide Time 16:53)



The “array *h*”

For each position *i* in pattern *P*, define h_i to be the length of the longest proper suffix of $P[1,\ldots,i]$ that matches a prefix of *P* and $P(h_i+1) \neq P(i+1)$.

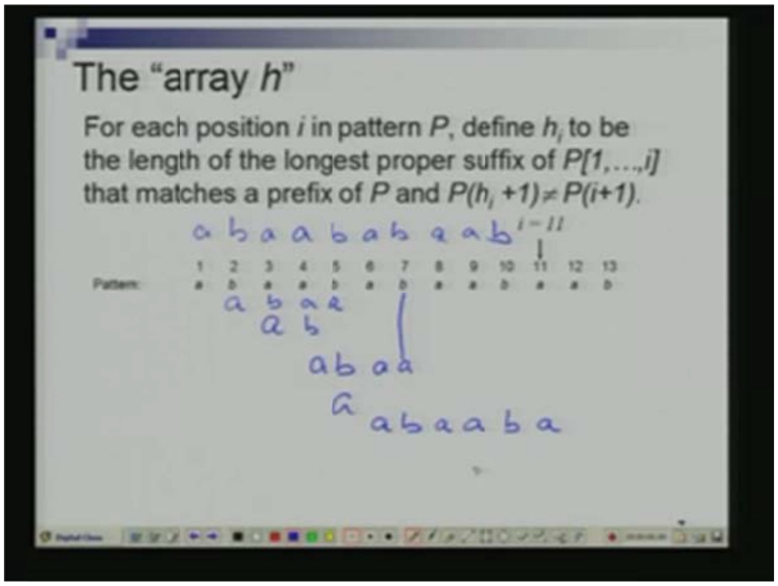
	1	2	3	4	5	6	7	8	9	10	11	12	13
Pattern:	a	b	a	a	b	a	b	a	a	b	a	a	b

7

What is the information ‘hi’? hi is the extent to which I will shift the pattern, if there is a mismatch at position i+1, that’s going to be the semantic. So we will understand this. How much I am going to shift the pattern right we will of course try to make the smallest possible shift. We will come to all of that. So we are going to make a certain amount of shift. We will see what the right shift is. Smallest or largest because it’s both smallest and largest. Smallest so that you don’t miss out any pattern. Largest so that you don’t waste any comparisons. So it’s both smallest and largest. So you can pick. Suppose this is what is happening, this is my pattern, I am now not showing you the text at all. But I tell you just the following information that the mismatch happened at position 11. So you don’t know what the text is but the mismatch happened at position 11.

How much should you shift the pattern by? Let’s draw a picture and let’s see how much should you shift the pattern by. Let’s say I shifted the pattern by one unit. Then what will happen? I will have a b a a. This is useless. I am now comparing the pattern against the pattern itself. Why because you know that text is the same here. That was I said I don’t know what the text is but I know now what the text is. It has to be this. This is one key observation. So now if I shifted the pattern by one unit, then clearly it could not match against the text. So one is useless. Let’s shift it by two units. If I shift it by 2, I get a b. It’s of no use again. Let me draw the shift by one. Also a b a a. Useless again.

(Refer Slide Time: 20:54)



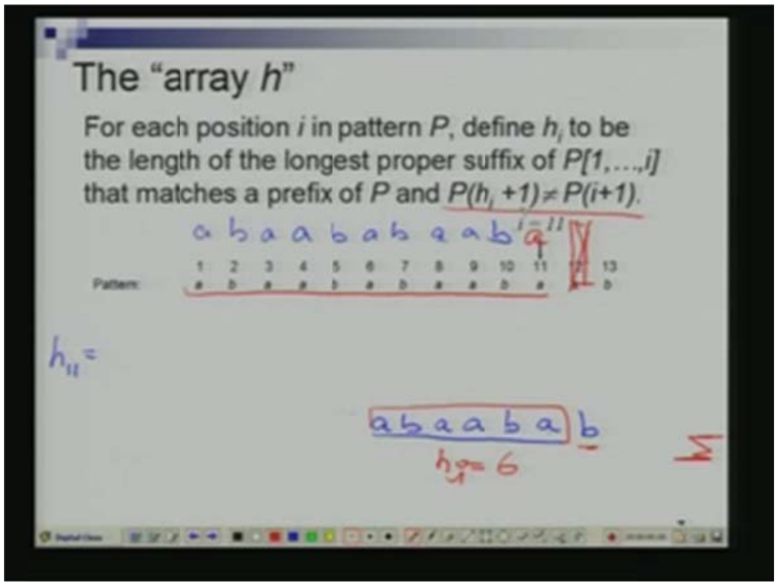
The "array *h*"

For each position *i* in pattern *P*, define h_i to be the length of the longest proper suffix of $P[1,\ldots,i]$ that matches a prefix of *P* and $P(h_i+1) \neq P(i+1)$.

	1	2	3	4	5	6	7	8	9	10	11	12	13
Pattern:	a	b	a	a	b	a	b	a	a	b	a	a	b

Shift by two? Useless because this is not matching this. Shift by 3. a b a a. Useless. Shift by 4? a. Useless. Shift by 5? So I can keep playing this game. a b a a a. We have shift by 5. (Hindi) So we are trying to shift by 6. But that is also useless. 7 and 8 are also useless. Actually I want to make i + 1 = l2. So 12 is the first place at which the mismatch occurs. 5th is okay. It was a b a a b a (Hindi) which is not the same as this. So this is acceptable because (Hindi). So we will say that 5 is the reasonable value to shift by. So we will come to what h_{11} or the 11th location of this array would be. So I have defined h_i to be the length of the longest proper suffix of P(1) through P(i) that matches the prefix of P. let's understand what this is. What is the suffix of P (1) through P (i)? Now recall we said this would always be an 'a' and there is a mismatch here (Refer Slide Time: 25:22).

(Refer slide time 27:45)



The "array h"

For each position i in pattern P, define h_i to be the length of the longest proper suffix of $P[1,\ldots,i]$ that matches a prefix of P and $P(h_i+1) \neq P(i+1)$.

a b a a b a b a a b a $i = 11$

Pattern:	1	2	3	4	5	6	7	8	9	10	11	12	13
	a	b	a	a	b	a	b	a	a	b	a	b	b

$h_{11} =$

$\boxed{a\,b\,a\,a\,b\,a}\;b$

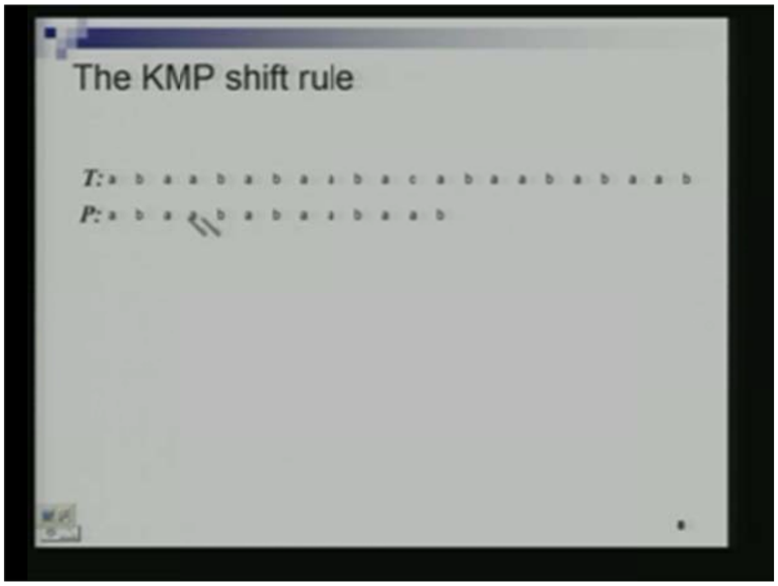
$h_{11} = 6$

So P (1) through P (i) is this part of the pattern (Refer Slide Time: 25:29). What is the suffix of the P (1) through P (i)? Prefix is contiguous parts before. So ‘a’ is a prefix. ‘ab b’ is a prefix. ‘ab a’ is a prefix. ‘ab a’ is a prefix and so on. What is a suffix? ‘a’ is a suffix. ‘ab b’ is a suffix. ‘ab a’ is a suffix. ‘a b a a b a’ is a suffix. So I am finding the longest proper suffix. Proper suffix means suffix which does not include the entire text. Like proper subset means strictly smaller. So that’s what proper suffix of P(1) through P(i) means that matches the prefix of P. So I have to find out length of the longest suffix of P(1) through P(i) that matches the prefix of P which is exactly this. This is the one we showed here. Why should the suffix match the prefix of P? Because when we shifted, it is the prefix of P which comes towards the suffix. I have to find out the longest match because that determines how many we have to shift without missing out on a possible match somewhere. So I have to find out the longest possible match and then there is the additional congestion here that the i+1th character, so ‘i’ is the length. I am finding out h_i - the length of the longest suffix which matches the prefix of P and then I want that (i + 1)th character of this should not be the same as the (h_i + 1) th character of the pattern because hi this (Refer Slide Time: 27:31). So h_i then is 1, 2, 3, 4, 5, 6.

So I don’t want that the 7th the same as P_{12} and why did we require this? Because we know that there is a mismatch happening here. So we want that better be different. So this is the definition for our h_i. So it’s a bit of a tricky definition but you understand what the argument is. So it’s not really how much you are shifting by. If the pattern has matched the first 11 positions, then h_{11} says that “okay, if there is a mismatch in the next position, then what is the shift?” so after the shift how many characters still continue to match? 6 characters and what is the extent of the shift? 11 – 6 = 5. The logic is very simple. You are trying to determine what is the extent to which you can shift the pattern. You are trying to determine what is the smallest extent by which you can shift the pattern. You will say smallest is one but smallest is not one because one is quite meaningless. You know that clearly there will be mismatch happening by shifting the pattern by one. So what is the smallest extent by which you can shift the pattern without you doing a necessary work? You know that there are certain mismatches happening. So you want to figure out the right extent to shift but you don’t want to shift by huge amount either. Why did I make shift by 11 units all together because then I would have missed out on certain possible matches.

Let’s continue. so this (Refer slide time 30:14) part of the pattern what we already discussed in the same as this one and 7th position is not the same as the 12th position which means that with the value of h_i - h 11 as 6, then this condition is true. This would be 7. The 7th position is not same as the 12th position. i is 11 and hi is 6. If there is no proper suffix of P (1) through i with this above property then what should the value of h (i) be? What is h (i) capturing? (Hindi). So let’s take some examples and then this will be clear. So this is my text (Refer slide time 31:35) and this is my pattern. Where is the mismatch happening? In ‘c’ or ‘a’.

(Refer slide time 31:35)



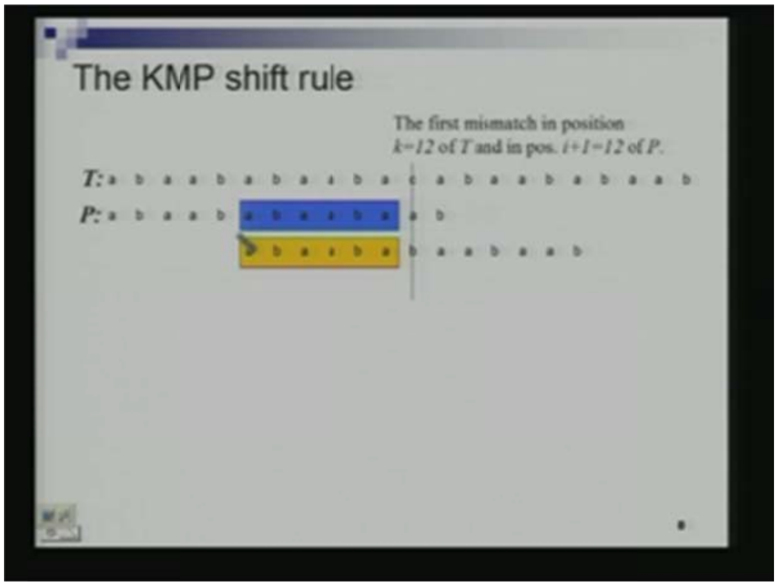
The KMP shift rule

T: a b a a b a b a a b a c a b a a b a b a a b

P: a b a a b a b a a b a a b

This is position 12 of the text. So till 11 positions, matches happen. So the value of i is 11 again. h_{11} is 6. By how many are we shifting? 5. So that’s what is happening. (Hindi). So last six characters and first six characters are the same. You can actually move the pattern all the way up to here and further this is different from this. So there is a potential this guy matches with this. It does not that there is a different thing. There is a potential that this guy matches with this. So we will shift it by so much amount. So this is the shifted pattern. h_{11} is 6. So you were shifting P to the right so that P (1) through hi - 1 through 6 aligns with this as the suffix of text. So k is the position of mismatch. k -1 is before this and k - hi is this part. So it aligns with this perfectly.

(Refer Slide Time 32:14)



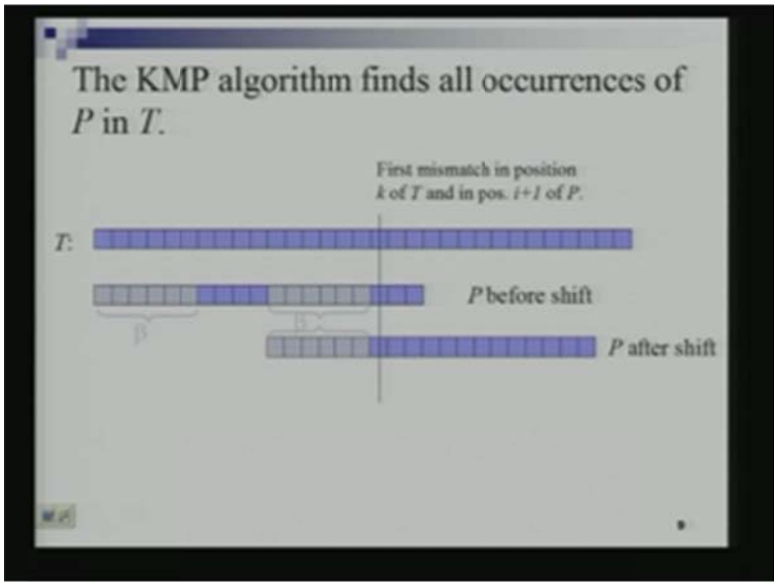
The KMP shift rule

The first mismatch in position $k=12$ of T and in pos. $i+1=12$ of P.

T: a b a a b a b a a b a c a b a a b a b a a b
P: a b a a b a b a a b a a b
a b a a b a b a a b a a b

These parts should match. Why? Because this is the same as this which is the same as this. So they match. So in other words we are shifting (P i – hi) places to the right. Suppose the pattern was only up to here, a b will not be the part of the pattern. So that means that till the 11th position everything is fine. How much should I shift forward by? Again the same i - hi because by doing that, we would not be missing out on any other patterns then. So everyone understands why we are looking at longest such matching part? So that minimum shift occurs or maximum shift occurs? So that minimum shift occurs. That way the minimum amount of shift is happening. But why did we want a minimum amount? We want the largest possible shift. So the right answer should not be so that minimum shift occurs. It is so that we don't miss out any patterns. That's why we are looking at the longest prefix. If I were to take shorter prefix, I might end up missing out on some guys. Why does this work? Why is it that we are not missing out on any pattern by doing this? We have to argue correctness of the algorithm.

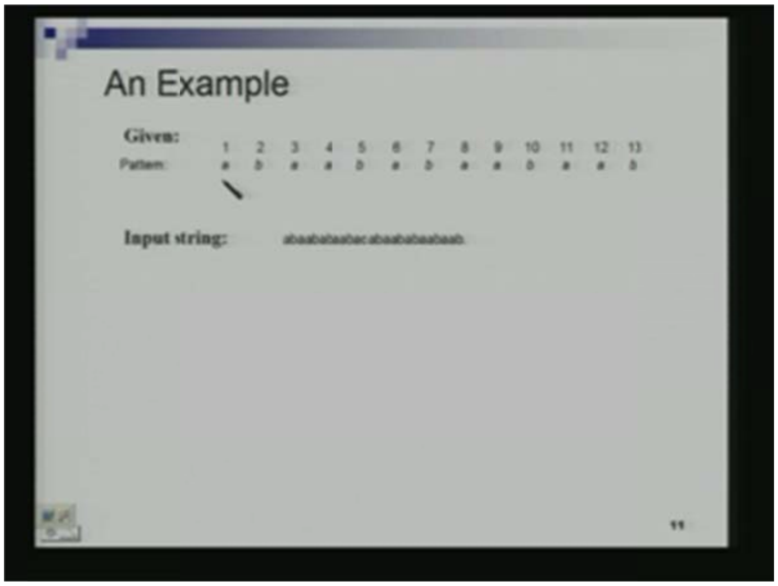
(Refer Slide Time 35:19)


The KMP algorithm finds all occurrences of *P* in *T*.

So this is my piece of text. The mismatch was at position k of that text and position I + 1 of the pattern. This is position I + 1 of the pattern this is the position k of the text (Refer Slide Time: 35:43). In this case k and I + 1 are turning out to be the same but there could be in some part of the text even before this case even before this I + 1 and k would not be the same. This is before the shift and these 6 character are same as this character which is why we could move it this way. So this is the situation after the shift. So if this part is beta, this substring is also the same as beta. Suppose this was not the case. That is, the KMP algorithm missed out on certain patterns, they were certain patterns in the text which we could not find the procedure at all, we will argue that certain things cannot happen.

Suppose there is this missed occurrence of the pattern, (Hindi), these are the same as this. But we missed out on this one. This guy mismatched only at this. This better be the same as this. So this is also alpha. So these alpha characters, this prefix of the string of length alpha matches the suffix of P 1 through Pi of length alpha and alpha is more than beta. So this violates our definition. Alpha is more than beta which is hi. These are longer. These matches this. These is a longer prefix. Yes agreed. But this character is also the same as this. (Hindi). So P l = tk because they matched but tk is not equal to pi plus one because there was a mismatch. So pl is not equal to pi + 1.
So it could not be that this and this are the same for which reason we did not take this longer suffix or prefix. So it is also the case this and this are the different. There was one requirement that we are looking for the longest prefix which matches the suffix of P 1 through P i and P i +1 is not equal to P of hi + 1. Both conditions need to be satisfied. (Hindi). Here hi value was not correct and this establishes the correctness of the algorithm. So if you have the correct value of hi, these things would work fine. So let’s take an example. This would now be clear after this. I have a certain (Refer slide time 40:16) pattern which is 13 character long. This is the input string.

(Refer Slide Time: 40:25)



An Example

Given:

	1	2	3	4	5	6	7	8	9	10	11	12	13
Pattern:	a	b	a	a	b	a	b	a	a	b	a	a	b

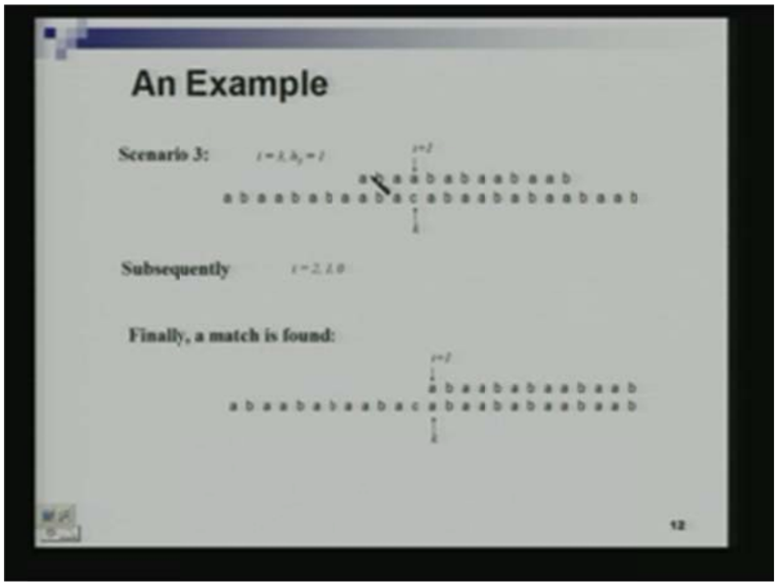
Input string: abaababaabacabaababaabaab

11

We will expand it out. This is the array h. Suppose I have calculated it in some manner, array h can be calculated for every I with some amount of effort. You can figure out what h of I should be. We will see how to do that also. Suppose I have figured this out, for instance this is the same pattern, at 11 you have 6. Let's see how we are going to use this information. Let's do a complete example. So this is my pattern. This is the text. The first mismatch occurs at position 12 which means till position 11 there was a match. So I go to position 11 in the array. I see the 6 written there. There was a 6 written in position 11. So what does it mean? I have to shift by 11 - 6 = 5 units. This 6 is telling me how much of the pattern is going to remain matched after the shift. So 6 characters of the pattern will remain matched. (Hindi). h_{11} is 6. So I - hi is 11- 6 which is 5. So I have to shift by 5 units. As you can see, the pattern has been shifted by 5 units.

The pattern is the one on top and the text is below. Now again I will continue to look at this position. Note that I don't have to try and match the first 6 positions because they are now already matched by the definition of the function h. They are matched. But this 12th position (Hindi). But what position in the pattern is it? This is position 7 in the pattern. The value of I is 6. Now I go to 6 and I see. So h_6 is 3. So now I am going to shift by I - h_6. 6 - h_6 = 3 units. I shift by additional 3 units and this is the situation I have now (Hindi) which we have to shift by 2 units.

(Refer Slide Time 43:15)



An Example

Scenario 3: $i = 3, h_3 = 1$

```
                  a b a a b a b a a b a a b
a b a a b a b a a b a c a b a a b a b a a b a a b
```

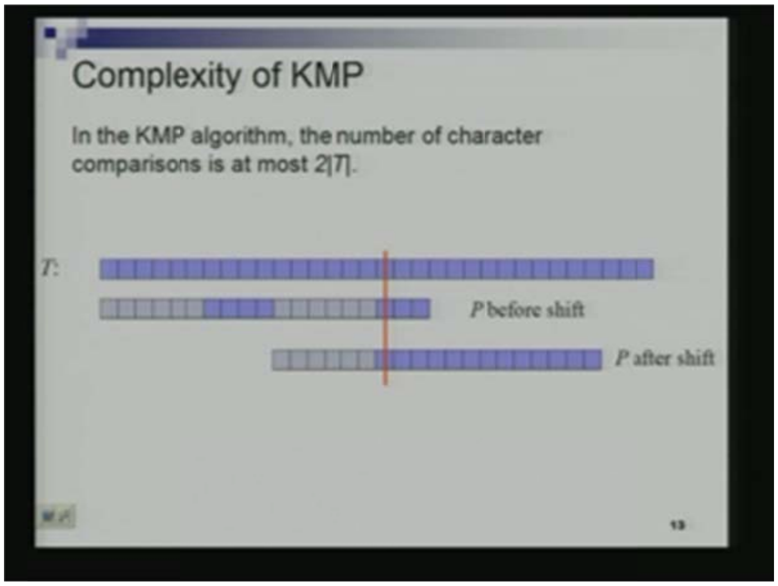
Subsequently $i = 2, 1, 0$

Finally, a match is found:

```
                        a b a a b a b a a b a a b
a b a a b a b a a b a c a b a a b a b a a b a a b
```

So we shift 2 units to the right and so we will get a b after the shift we will get a b and something but b and c still mismatch which means we will now shift by 1 unit. a and c mismatch. So we will shift by another 1 unit (Hindi). When you shifted beyond the c, then there was the match from when you found the match. So if you had the h function, you can use that to determine how much you should shift by at every step so that the match can be found and this shift function is designed in such a manner that you will never miss out on any potential match. Every one followed this 210 business? We come to calculation of h. How much time does our algorithm take?

(Refer Slide Time 45:23)



Complexity of KMP

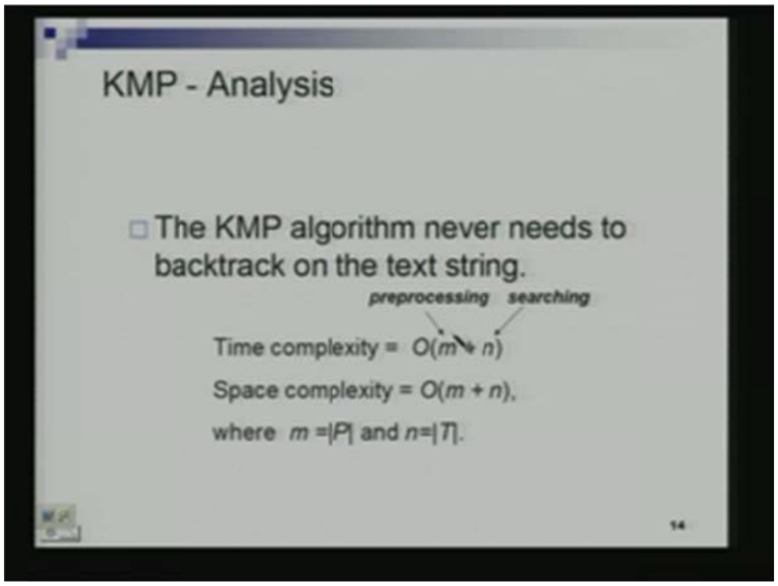
In the KMP algorithm, the number of character comparisons is at most 2|*T*|.

Suppose h is given. We will see how to do the h. (Hindi). We will count the number of comparisons. (Hindi) this was the first step at which we had the mismatch. (Hindi) because it did not result in the match. (Hindi) every time we were comparing with c. So how many times would we be comparing with a certain character? Many times up to the length of the pattern. But that doesn't sound very good because that would mean that the number of comparisons is huge. For each character, the maximum number of comparisons could be large. But let's look at the total number of comparisons put together. So let's call this unsuccessful comparisons. What is an unsuccessful comparison? Comparisons that result in a mismatch. Now how many unsuccessful comparisons are there, over all put together? (Hindi) every time I do an unsuccessful comparison, I shift the pattern. (Hindi) at most, the number of the size of the text types. So the number of unsuccessful comparisons is at most the size of the text which is 'n'. So unsuccessful comparison are not too many.

How many successful comparisons are there? (Hindi) these were all the successful comparisons. After I shift, these characters are a part of successful comparisons. We will never compare against these characters again. (Hindi) we will never compare this part of the pattern; the part before this red line with the text. So each character of the text which is ever part of the successful comparison is compared only once. There's no back track happening. The number of successful comparison is at most the size of the text. The total number of comparisons is the number of shifts plus the number of unsuccessful comparisons. So we broke up our comparisons into two sets: successful and unsuccessful.

Once a character is part of the successful comparison it's never compare again. If a character is part of the unsuccessful comparison, it might be compared again. But it results in a shift. So every unsuccessful comparison results in a shift. So total number of shifts since it can be no more than the size of the text. No two successful comparisons can be more than the size of the text. So total number of comparisons therefore it at most two times the size the text. So total number of comparison that are required is the size of the text.

(Refer Slide Time 50:48)



KMP - Analysis

- The KMP algorithm never needs to backtrack on the text string.

Time complexity = $O(m + n)$

preprocessing
searching

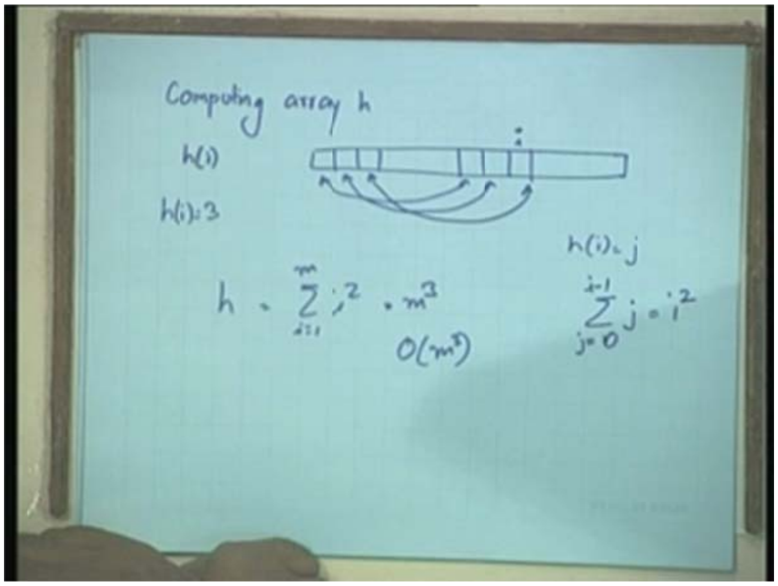
Space complexity = $O(m + n)$,

where $m = |P|$ and $n = |T|$.

14

But what about computing the array h? How do we compute the various hi’s and how much time does it take? So we are going to call that preprocessing time. So recall that the size of the pattern was n. What one can argue is that the time required to compute the array h is only order m - the size of the pattern and the time required to do the searching is order n - the size of the text. So the total time therefore is order (m + n). I have not told you this as yet: ‘y’ is computing the array h. ‘y’ can we done in order m time. Let’s see why. So we have to compute array h.

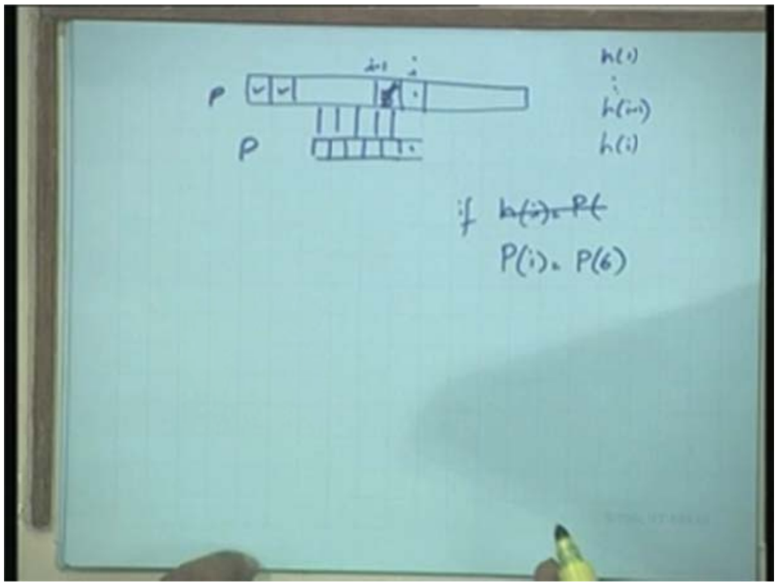
(Refer Slide Time 55:09)



What was our definition of an array h? Suppose I have to compute h (i), I look at my pattern. I was certain pattern. This is the ith position. What is h of I, who can remain me what h of I is? The longest proper prefix of the pattern which is also a suffix of 1 through I and that next character should be different [Hindi]. So hi is the value, it’s the longest suffix, longest prefix or longest suffix whatever longest prefix of the pattern. So what are the potential value of hi, what are the different value hi can take? zero to m, zero to I -1 because we also said proper. What is the proper subset? When is the subset proper, when it’s not equal to the set. So that’s what the proper suffixes, when it’s not in the entire string.

So essentially it can take i different values, give or take a one and then I can try out all those various values. What is that mean? Suppose I am trying out the value h (i) =3. What should I do to try this out? I should look out the last three character here and the first three character and see if the other same. If this is the same as this, if this as the same as this and if this as the same as this then h (i) can be 3, hi is more than or equal to 3 (Refer Slide Time: 53:43). So I will try out 3, 4, 5, 6, 7 all the way up to i-1 and whichever are the possible value I just take the largest among them. So how much time does this take? So how much time do I take to compute hi, I will try out all the i different values. I will do all the so many comparison, so in particular if I choose a value of h (i) equals j then how time do I need to do check whether j is the candidate value? j time. I will take j, j going from 1 through i-1, zero through i-1, so this is i square roughly. So for h (i) I will take i square time, so for entire array h, I will take summation i square I going from 1 through m which is m cube time, huge amount of time. This should be o of m cube; divide by what? (Refer Slide Time: 55:11) We just taking computing the total time (Refer Slide Time: 55:26) huge amount of time right. So we want to do something better. We can use the previous value of h (i) which have been calculated how can we use that? So that's a nice idea in fact that's the right idea. So to compute h (i) so we are going to assume the following that when you are trying to compute h (i) you already have the previous values h 1 through h (i) -1. These are already there.

(Refer slide time 1:00:00)



Note that in doing this computation of array h we don't require the text at all. It's being done just on the pattern. You can take your pattern, compute your h and then you can match it again which ever text you want. So the text can be whatever you want, once you computed this then you can just go head and work with the text. Suppose I am trying to compute h (i) again and I know all the values of h 1 through h (i) -1. How can I use these to compute h (i)? Who can tell me? Span of the stock using the stack. Now what can we do? I don't think that is the same idea but can someone tell me what should I do here?

So for h (i) -1 we know so if h (i) -1 was let's say 5, so we know that the first five match with these 5. So these 5 match with these. So these are the first 5 of the pattern, these is the pattern p these also the pattern p. So the first 5 of the pattern match with i-1, i-2, i-3 basically i-5 to i-1 of the text of this pattern itself. Now how do I know what h (i) is? These doesn't solve the h (i) problem. Now if this 6 character of the pattern matches this then can I say that h (i) is equal to 6. Can I say that? Why not? So I claim the following, if p (i) = p (6), the 6 character of the pattern. [Student: it is atleast h (i-1) so we will try of z assuming that it is at least five, so that's what we have to do we have to do something like (Refer Slide Time: 59:56). There was zeros also so it need not be an increase, no but it need not be equal to h (i-1). It could be zero also. We are saying but it could potentially if this and this match now, I am sorry I am confusing my array h with my (Refer Slide Time: 1:00:23). Student: h (i+1) is not equal to h (i+1). Staff: so they would not match so it has to be less than 5, so they would be not match. Student: so it has to be less than five greater than six or. Staff: okay so that's a good question we will take this off in the next class. So today we looked at pattern matching in a text. So we looked at very simple algorithm for doing that which had an order mn time then we looked at the knuth-morris-pratt algorithm which we argued has the running time of order m + n. So that was the improvement, we went from a mn to m + n this is the modulo fact that you can compute the array h in order m time. So we shown you how to compute the array h and order m cube time and we are going to see how to do that in order m time in the next class.