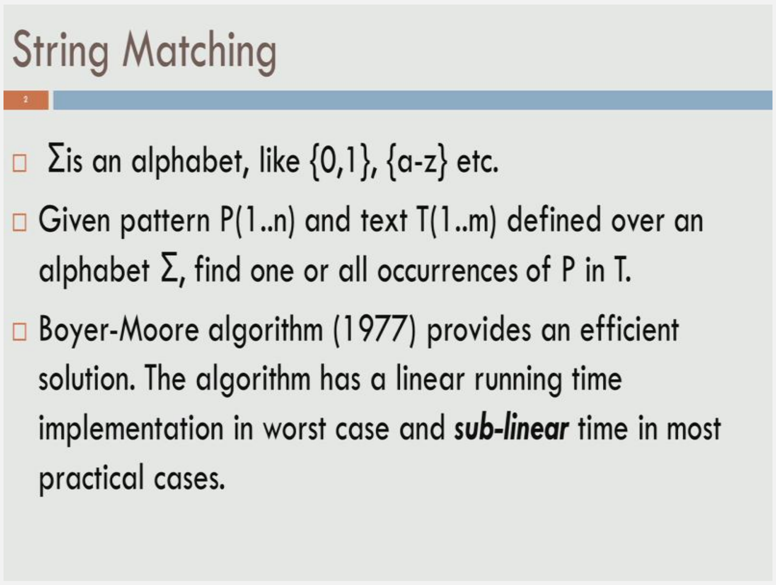
Programming Data Structures and Algorithms

Prof. N.S.Narayanaswamy
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 10
String Matching - Boyer Moore Algorithm

Hello every one, welcome to this lecture number 10 on algorithms for the string matching problem. String matching problems are almost everywhere ubiquitous every computing system provides us with facilities to match strings, browsers, editors, shells all of them provide us with abilities to, with utilities to match strings. And one of the algorithms that is of interest and which is a very efficient algorithm is called Boyer Moore algorithm.

(Refer Slide Time: 00:52)



String Matching

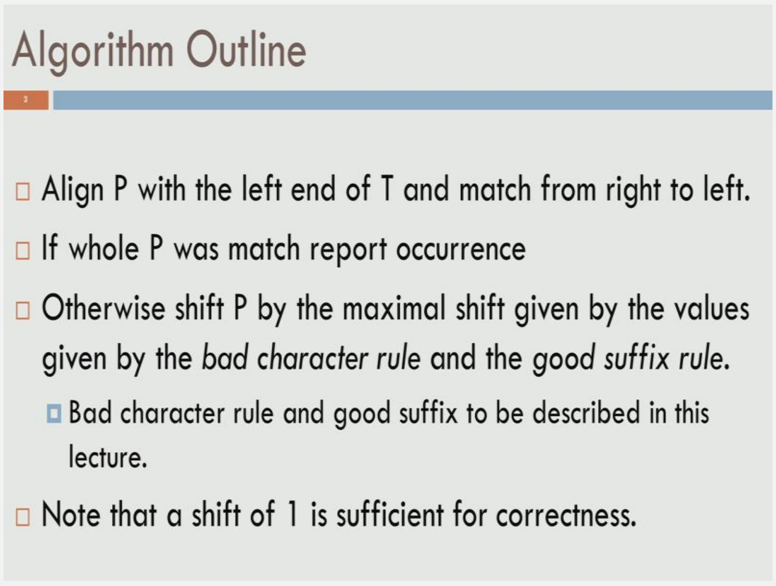
- $\Sigma$is an alphabet, like {0,1}, {a-z} etc.
- Given pattern P(1..n) and text T(1..m) defined over an alphabet Σ, find one or all occurrences of P in T.
- Boyer-Moore algorithm (1977) provides an efficient solution. The algorithm has a linear running time implementation in worst case and ***sub-linear*** time in most practical cases.

This algorithm was invented in 1977 and specifically it has an implementation which runs in linear time in the worst case and one can actually do an analysis and argue that it takes sub linear time to find a pattern match in most practical cases. So, let us get to the formal details of the string matching problem. In this problem, we have sigma an alphabet. Example of an alphabet is a two letter alphabet consisting of the letters 0 and 1. This is the alphabet over which the binary strings are defined.

One can also think of the alphabet which consists of the letters a to z, the 26 letters of the English alphabet and. So, on and. So, forth. The string matching question is the following. You are given a pattern of m letters and this we refer to as the pattern P and the characters in that pattern or the letters in the pattern are index using indexes from the range 1 to n. And we are given a target text which is T which has m characters in it or m letters in it and both P and T are over the same alphabet.

Now, the question is to design an algorithm to find whether P occurs in T that is one question. The other question is to find all occurrences of P and T. So, the Boyer Moore algorithm is a very interesting algorithm which uses, quite a lot of concepts from data structures to design an efficient algorithm for this problem. As we go along the description of the Boyer Moore algorithm, we will see what is straight forward and a simple algorithm will do, as a special case of the description of the Boyer Moore algorithm.

(Refer Slide Time: 02:49)



Algorithm Outline

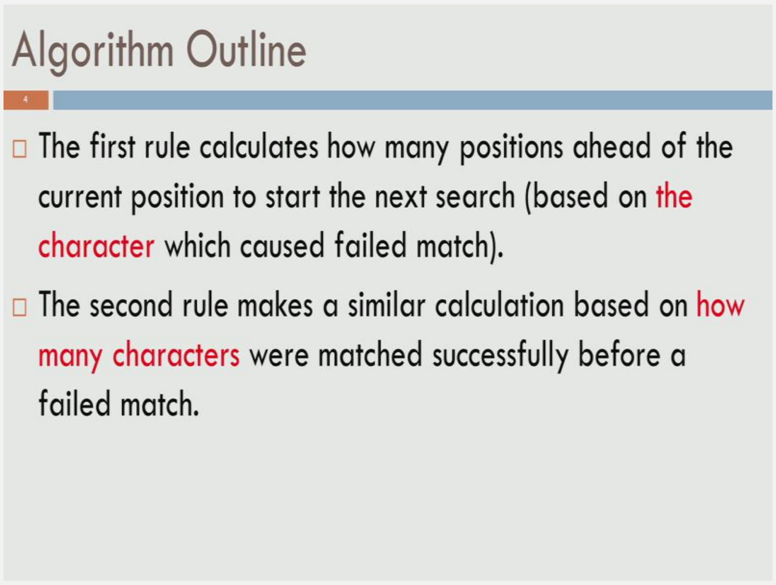
- Align P with the left end of T and match from right to left.
- If whole P was match report occurrence
- Otherwise shift P by the maximal shift given by the values given by the *bad character rule* and the *good suffix rule*.
  - Bad character rule and good suffix to be described in this lecture.
- Note that a shift of 1 is sufficient for correctness.

So, the outline of the algorithm is the following. We look at the pattern P and look at the given text T, now we align P with a left end of T and we check for a match from the right end of P towards the left end of P. We will discuss this more detail in the next slide, but I hope this picture is clear. If P is found to match the sub string of T with which it is align, then we report that P has been found, otherwise we identify how much to shift P to the right or rather to align P with another sub string of T which occurs to the right hand side of the text string T and this maximal shift that we used is given by one of two rules which are called the bad character rule and the good suffix rule.

And this lecture is to basically describe to you the details of the bad character rule and the good suffix rule and also show you an implementation of it. The running time analysis is not done in this lecture, on the other hand the formal implementation is discussed. So, here we get an idea saying that if P does not match with the current alignment with T, then the natural thing that we would do is, to shift P one cell to the right and align it with that part of T and then iterate this procedure, clearly we get an algorithm which will check if P is present in T or not.

It is a very simple exercise to analyze the running time of this algorithm. You can see that it is the order of m times n, because P has n letters in it. It is matched against the sub string of T which also has length n. Therefore, an order n time we can check if P matches with a sub string of T with which it is aligned and P has to be shifted an order of m times to the right in the worst case. Therefore, this is an order m n algorithm. We will see that by observing a lot of structure with respect to P and T, we should be able to do better in the worst case.

(Refer Slide Time: 05:19)

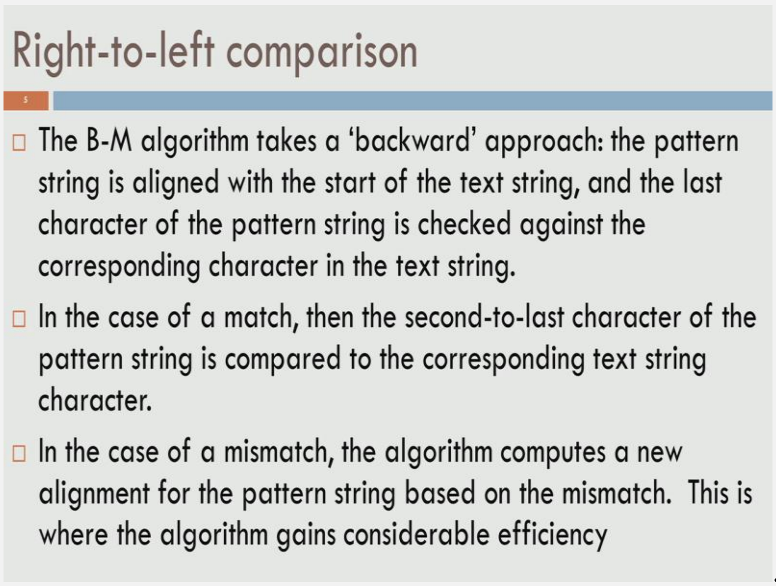


Algorithm Outline

- The first rule calculates how many positions ahead of the current position to start the next search (based on the character which caused failed match).
- The second rule makes a similar calculation based on how many characters were matched successfully before a failed match.

So, let us just discuss this shifting. Of the two rules that we just spoke about, the first rule tells us how many positions the pattern P should be moved to the right for the next pattern matching exercise or for the next search. Now, this depends upon the character with which our match was failed. The second rule also tells us, how many positions to the right the pattern P has to be moved and this is dependent upon the number of characters which were matched successfully before a failed match was obtained. We will see examples in the coming slides.

(Refer Slide Time: 06:04)



Right-to-left comparison

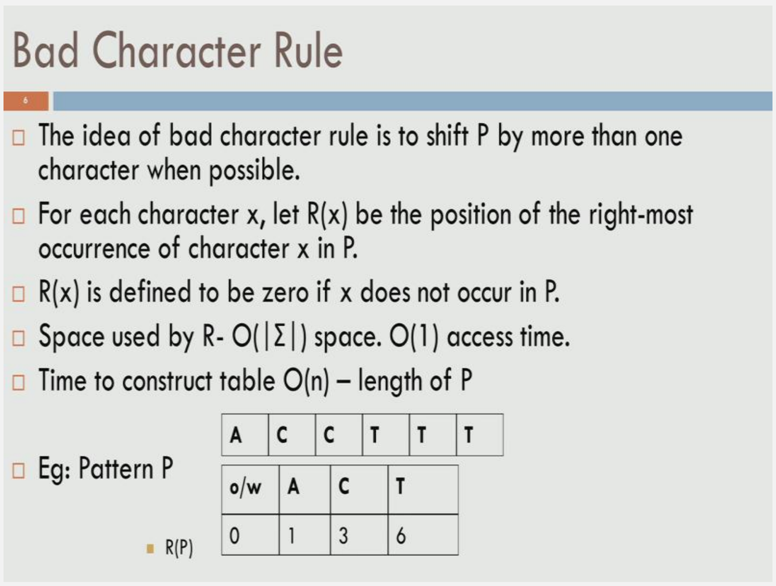
- The B-M algorithm takes a 'backward' approach: the pattern string is aligned with the start of the text string, and the last character of the pattern string is checked against the corresponding character in the text string.
- In the case of a match, then the second-to-last character of the pattern string is compared to the corresponding text string character.
- In the case of a mismatch, the algorithm computes a new alignment for the pattern string based on the mismatch. This is where the algorithm gains considerable efficiency

.

So, let us just understand one of the central features of this Boyer Moore algorithm which is the right to left comparison. So, the right to left comparison is very simple, imagine that the pattern P is matched against the text string T at some place. So, what you do is, you check if the last character of P matches with the corresponding location in T. If there is a match we go to the second last character and then perform a comparison.

This comparison process goes from right to left, we start with the last, then the second to last and third to last and. So, on and. So, forth, till we come to the starting string of the pattern P, staring character of the pattern P. Whenever we find the mismatch, our algorithm, these two rules will tell us how much to shift the pattern P and work with a new alignment and repeat this right to left comparison procedure. So, this is essentially the approach of one of the, this is one of the essential things in the Boyer Moore algorithm.

(Refer Slide Time: 07:16)



Bad Character Rule

- The idea of bad character rule is to shift P by more than one character when possible.
- For each character x, let R(x) be the position of the right-most occurrence of character x in P.
- R(x) is defined to be zero if x does not occur in P.
- Space used by R- O(|Σ|) space. O(1) access time.
- Time to construct table O(n) – length of P
- Eg: Pattern P

A	C	C	T	T	T

  - R(P)

o/w	A	C	T
0	1	3	6

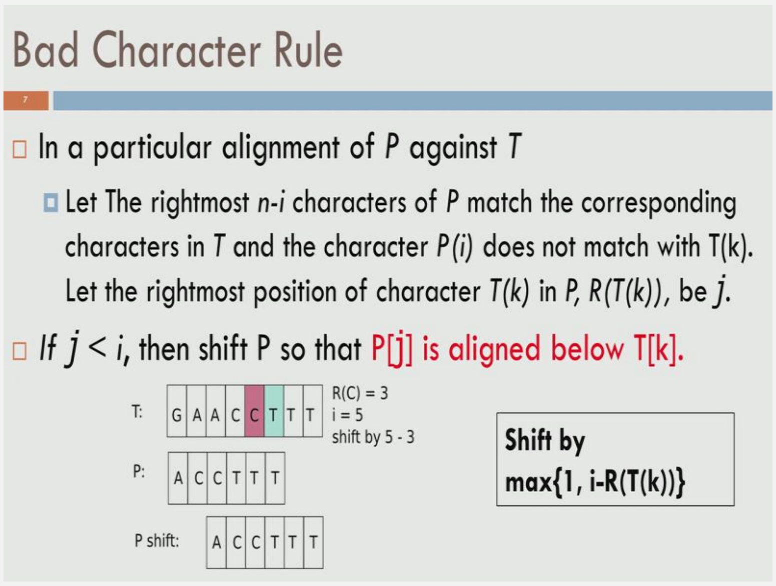
So, let us just go to the bad character rule, recall that this is the first of the two rules that we spoke about. In the bad character rule, the motivation is to come up with an approach. So, that the pattern P is shifted by more than one character whenever possible. To achieve this, an array R is maintained and the number of letters or the number of locations in this array is the size of the alphabet sigma. So, let us assume that x is a character in sigma and R of x takes the value zero, if x does not occur in the pattern P.

And if x does occur in the pattern P, R of x contains the index of the right most position in which x occurs in the pattern P. There is an example here, let us just take a look at this example. The pattern is the string A C C T T and T. Here, we show the array R of P, R is the name of the array and P is the pattern. So, the first one says that, if there is any other letter other than A C C T T T in the alphabet then R of P for that letter will take the value zero. This is the shortcut in the presentation.

On the other hand, A occurs exactly once and we keep the right most location where A occurs which is at the first location. Therefore, it takes the value 1. Observe that C occurs at the third location and it does not occur anywhere to the right of the third location, therefore, R of C is the value 3. T occurs at location 6 which is the length of the pattern string and therefore, R of T is the value 6.

This data structure as you can see can be populated in linear time, in one pass of the pattern array by looking at every character exactly once we can populate this array with the appropriate values. This is left as a small programming exercise which I am sure, all of you must be able to do.

(Refer Slide Time: 09:49)



Bad Character Rule

- In a particular alignment of *P* against *T*
  - Let The rightmost *n-i* characters of *P* match the corresponding characters in *T* and the character *P(i)* does not match with T(k). Let the rightmost position of character *T(k)* in *P*, *R(T(k))*, be *j*.
- If *j* < *i*, then shift P so that P[j] is aligned below T[k].

T:
G A A C C T T T
R(C) = 3
i = 5
shift by 5 - 3
P:
A C C T T T
P shift:
A C C T T T

Shift by max{1, i-R(T(k))}

Continuing with a bad character rule, let us see how to use this array R to come up with the appropriate shift of the pattern P whenever a mismatch is obtained. Let us present the rule first and then go to an example. So, let us consider a situation where P is aligned against T and for some i, the right most n minus i characters of P match the corresponding characters in T. And the character P of i that is the ith symbol in the pattern P does not match with a corresponding character in the text string T.

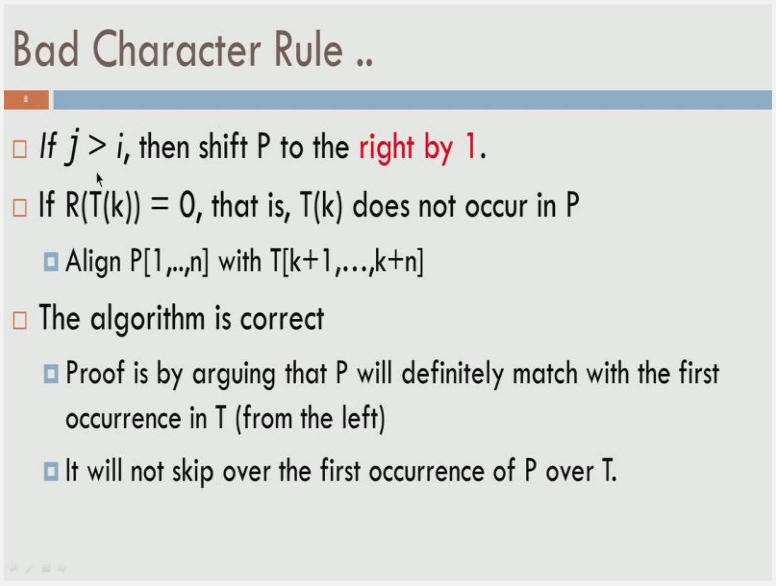
We look at this character T of k and identify the right most location where it occurs in the pattern. This recall is present in the array R let us refer to this particular value as j. Let me repeat this, T of k does not match with P of i and P of i and T of k are aligned with each other, R of T of k is the right most location in the pattern P where the character T of k occurs. Let us give a shortcut to this particular value and let us call it j. So, let us just go to this example and let us look at this alignment, this is the text string and P is aligned with it.

So, let us just do a comparison, as we do a right to left comparison of P with the match portions T or the aligned portions T, let us look at these two characters this is the match, then we come to the second to last character, this is the mismatch. Now, let us look at the cause for the mismatch. So, the cause for the mismatch is that in the text there is a C here and in that pattern, there is the T here, this is what we refer to as T of k in the general rule that we just note down and let us look at R of C.

Observe that R of C is the value 3 that is C occurs in the pattern, at the third location it does not occur anywhere else further to the right. Now, further let us also observe that the value j which is 3 in this case is strictly to the left of the place where the mismatches. The mismatch has happened at the index 5, 1, 2, 3, 4 and 5 the mismatches at the index 5, R of C is at location 3 and therefore, we shift the pattern P. So, that this right most occurrence of this mismatch symbol, when it occurs to the left is now aligned, this is the shifted symbol.

Observe that now, C and C are matched, this is the rule and the whole pattern P is shifted to the right and in this case as you can see, it is shifted by i minus R of T of k. Observe that i is 5 in this example and R of T of k that is R of C is the value 3 and therefore, it is shifted by two units, the pattern P is shifted by two units to the right. So, this is the bad character rule. One can observe that this is the correct option to perform, if indeed there is a match for P in T and if this match, observe that this is the correct operation, if P has to be matched to some sub string of T.

(Refer slide Time: 13:42)



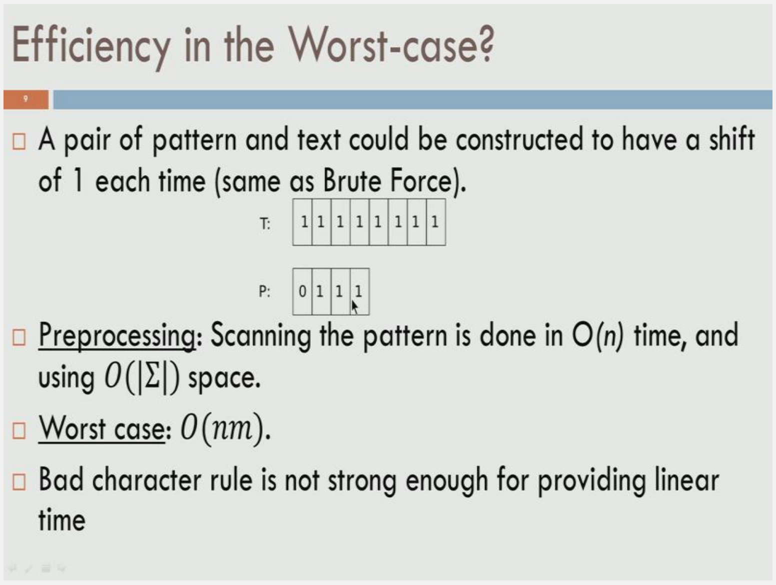
Bad Character Rule ..

- If $j > i$, then shift P to the right by 1.
- If R(T(k)) = 0, that is, T(k) does not occur in P
  - Align P[1,..,n] with T[k+1,…,k+n]
- The algorithm is correct
  - Proof is by arguing that P will definitely match with the first occurrence in T (from the left)
  - It will not skip over the first occurrence of P over T.

So, continuing further with a bad character rule, if j is more than i, in other words the right most occurrence of the mismatch character occurs to the point of mismatch, then shift P to the right by 1. There is also a third case here, if the mismatch character does not occur in P at all, then; obviously, the pattern P must be moved completely to the right of the current alignment string T that is align P of 1, 2 to n with T of k plus 1 to k plus n.

Recall that T of k was the place where there was a mismatch. It is easy to check back that the algorithm is correct, by checking that these three cases will ensure that P will definitely match with a first occurrence of T from the left, in other words it will not skip over the first occurrence of P in T the first occurrence is checked from the left hand side. So, therefore, the bad character rule is indeed a correct algorithm and therefore, what we have is a string matching algorithm, already. Let us now discuss how efficient is this algorithm.

(Refer Slide Time: 15:10)



Efficiency in the Worst-case?

- A pair of pattern and text could be constructed to have a shift of 1 each time (same as Brute Force).

T: | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

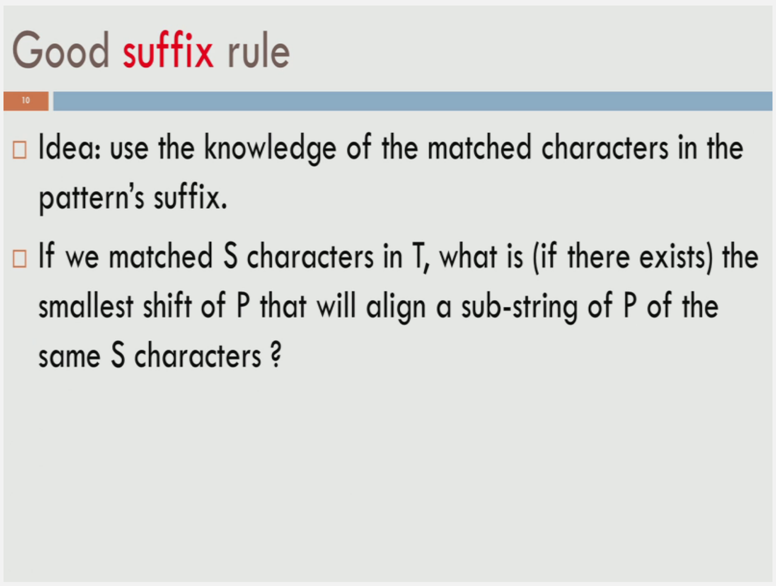
P: | 0 | 1 | 1 | 1 |

- Preprocessing: Scanning the pattern is done in $O(n)$ time, and using $O(|\Sigma|)$ space.
- Worst case: $O(nm)$.
- Bad character rule is not strong enough for providing linear time

Let us now check that this algorithm is extremely inefficient, let us consider the text pattern which is all 1’s followed by the pattern which is a 0 followed by three 1’s. Observe that for every alignment we will find a mismatch, because the pattern 0 1 1 is not present in the text string T that is very clear. However, if we applied the bad character rule, we will see that in this example the worst case time taken would be an order of n comma m, before we discover that this pattern is not present in the target at a given text string T.

Therefore, the bad character rule is not strong enough for providing the linear time algorithm, something else is necessary. So, therefore, let us just see what we need to handle. We have to handle the case when the mismatch, the right most occurrence of the mismatch character occurs to the right of the point at which the mismatch has been discovered. So, we need to identify the way of handling this particular case.

(Refer Slide Time: 16:30)

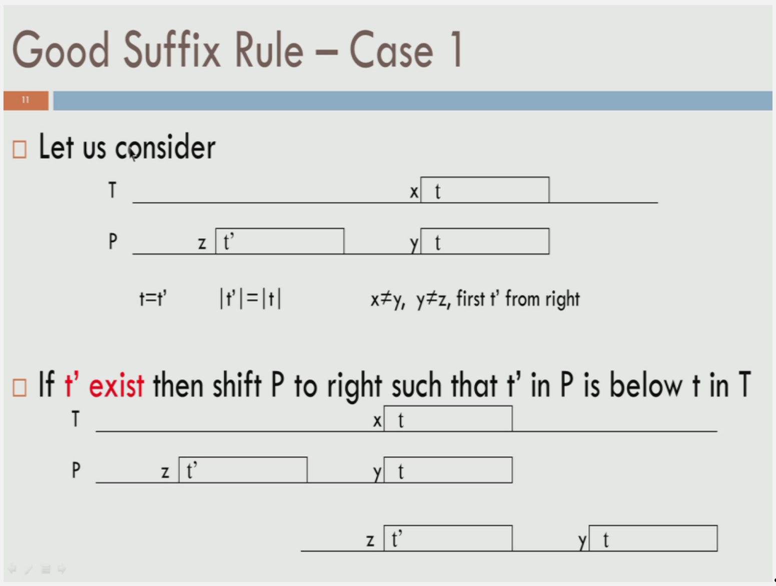


Good suffix rule

- Idea: use the knowledge of the matched characters in the pattern's suffix.
- If we matched S characters in T, what is (if there exists) the smallest shift of P that will align a sub-string of P of the same S characters ?

To do this let us just think of an idea and the idea would be to how to use that, how to use the fact that we have matched some characters in the patterns suffix with the current align portion of the text string T. How does one use this knowledge? So, let us just ask the question if we have matched a suffix of characters in T, now let us ask what is the smallest shift of P which will align a sub string of P with same suffix of characters, let us we visualize this pictorially in the coming slide.

(Refer Slide Time: 17:15)



Good Suffix Rule – Case 1

- Let us consider
- If t' exist then shift P to right such that t' in P is below t in T

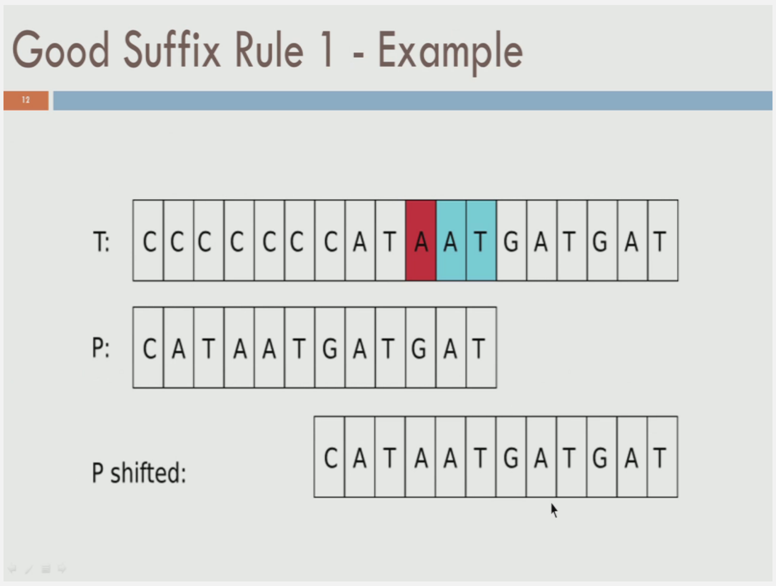
.

And this rule is called a Good Suffix rule and you will see why it is called a suffix rule. So, here is a target string T and the pattern which has matched is referred to as small t and here is the mismatch that is x and y are not the same. As we come from the right, we compare letter by letter and. So, many matches have been discovered between T and P and now x and y are different. Now, you need to ask how much should we shift, the pattern by.

So, let us see what would be a natural thing to do. If t occurs again has t prime to the left of t and let us assume that this is the first occurrence of t prime as we go from the right to the left. Then, the natural shift would be to shift P. So, that t prime matches with this t as it is displayed in this particular pictures. So, the shift for P would be shifted. So, that t prime aligns with t and this is a very important thing to do that if there is going to be a successful match that involves this t.

Then, such a successful match could only involve t prime and no other sub string to the right of t prime, because this is the first occurrence of t to the left of, t prime is the first copy of t that occurs to the left of this pattern t. So, therefore, this is the natural shift to perform, we will formalize this shortly, but before that let us take a look at an example.

(Refer Slide Time: 19:02)



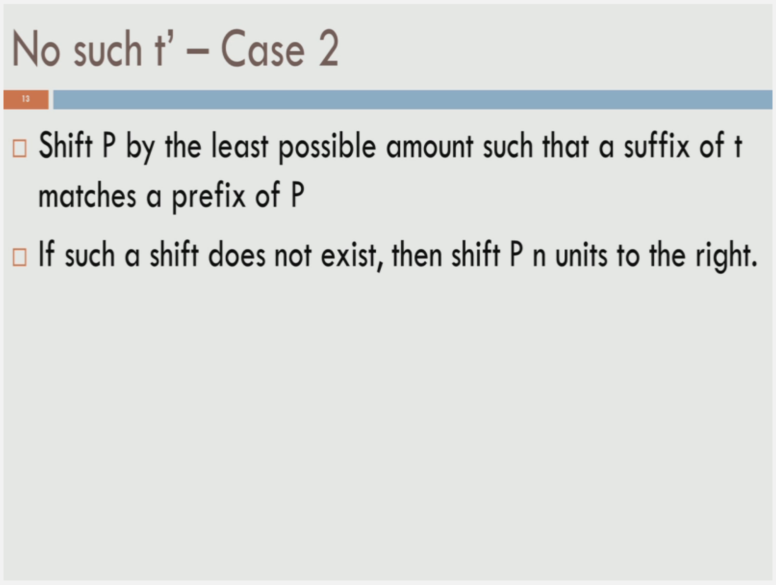
Good Suffix Rule 1 - Example

12
T: C C C C C C C A T A A T G A T G A T
P: C A T A A T G A T G A T
P shifted: C A T A A T G A T G A T

So, let us just look at the pattern and T aligned in this fashion, these two match A and A match, now there is a mismatch and A is this character. So, now if we ask how much should we shift P by, let us just check that there is an A T here, there is an A T here. So, of course, this A T could be aligned with this A T. So, that could be a shift of a certain amount. But, observe that the preceding symbol of this sub string is G and it is the same G which is the mismatch.

Therefore, we do not shift it. So, that this A T is aligned with A T, this A T. Let us look at this A T now and let us look at the preceding string A and in this case of course, there is a match and therefore, we shifted all the way, shift p all the ways. So, that this A T aligns with this A T. The specific property is that we look for a letter here which is different from the character, which cause the mismatch with A. In this case of course, it turns out that A and A are one and the same and we are actually, if you see this shift will indeed, check that there is a match of P in T. So, let us just formally state the first case of the good suffix rule.

(Refer Slide Time: 20:40)



No such t' – Case 2

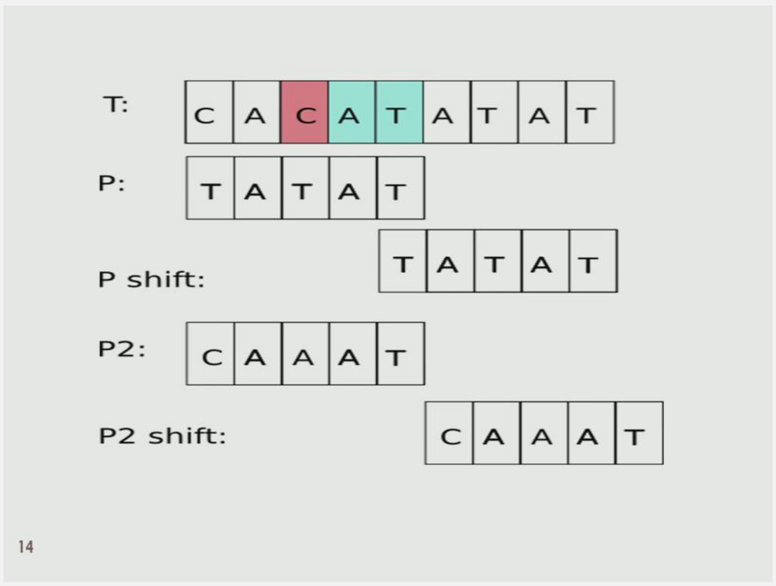
13

- Shift P by the least possible amount such that a suffix of t matches a prefix of P
- If such a shift does not exist, then shift P n units to the right.

Before we do that let us also check what happens when such a t prime does not exist. So, let us just go back to the slide and take a look at ((Refer Time: 20:52)) what we mean by this. If such a t prime does not exist at all, then how much to be shift P y. This is given by the suffix rule in the second case, using the second case. This is what we title the slide as, we call it if there is no such t prime then we are in the case two.

The idea now is to shift P the pattern by the least possible amount such that the suffix of t matches the prefix of P. One can check that this is indeed a correct step to perform and we will not miss any pattern, any occurrence of pattern P in t by this approach, indeed this is the minimal shift that must be done. Now, if such a suffix does not exist, then we recommended, you shift P n units to the right. So, these are the three steps in the good suffix rule algorithm.

(Refer Slide Time: 21:57)



Let us just take a look at an example for the case two. Recall that we have already seen an example for case one. There are two cases here, two examples here, one two examples if I… The first case of case two and the second one to illustrate the second possibility in case two. So, this is the target string and this is the text string, like before A T is a match and there is a mismatch here. Now, observe that there is no occurrence of A T to the left here, with the property that the preceding symbol is different from this particular T.

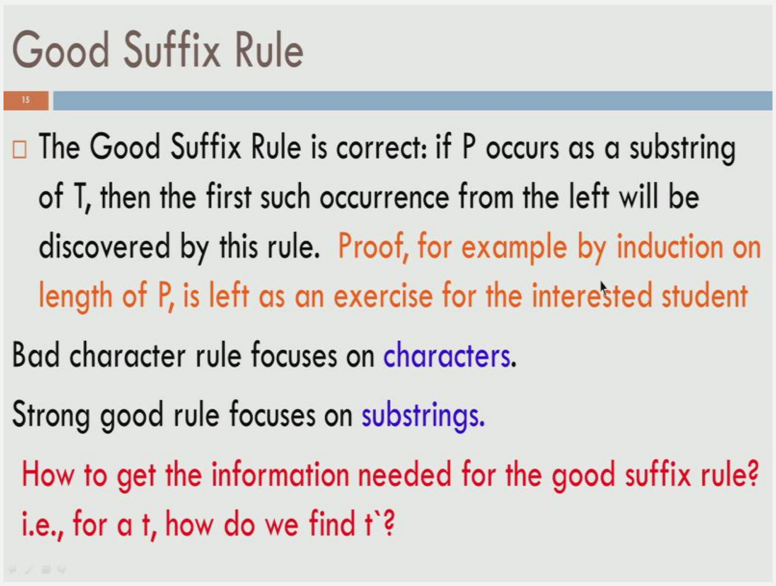
So, let us just take a look at this. This is A T it did match, this is also A T it did match, the preceding symbol of this is T same as the preceding symbol of the matched portion. Therefore, it is a bad idea to or it is an unnecessary effort to shift P to just ensure that this match happens and then perform a comparison again. Because, we already know that such a comparison will fail. Therefore, the idea now is to shift P. So, that a prefix of P matches with the suffix of T and we perform the minimum such shift and let us just see this.

The minimum such shift such that some prefix of P. So, let us look at the prefixes of P, that is T, T A, T A T and. So, on. What is the minimum shift of P such that, the prefix of P matches with a suffix of this particular part which we have refer to as T. In this case, we can see that this is the minimum shift that is possible. So, this is the first possibility in case two, the second possibility is that even such a shift does not occur. Then, we would have to shift the whole pattern n units to the right. In this case, it is five units to the right.

Let us just see such an example, the pattern in this case is the string C A A A T and the target is the same thing. Observe that this A T and this A T match, there is a mismatch between C and A here and now observe that the pattern A T here does not occur anywhere to the right and therefore, we would attempt to apply the first possibility of case two. But, observe that in the first possibility of case two, no prefix matches with a suffix of the matched portion already which is T.Because, all of them start with a C and C does not occur in this part at all.

Therefore, the algorithm suggest that simply shift P n units to the right which is shifted all the way to six units to the right and now align it with the target string T. So, this example simplifies the good suffix rule or this illustrates the good suffix rule, completely. So, as we have done. So, for in this course, it is very important to argue the correctness of every step that we performed.

(Refer Slide Time: 25:25)



Good Suffix Rule

15

- The Good Suffix Rule is correct: if P occurs as a substring of T, then the first such occurrence from the left will be discovered by this rule. Proof, for example by induction on length of P, is left as an exercise for the interested student

Bad character rule focuses on characters.

Strong good rule focuses on substrings.

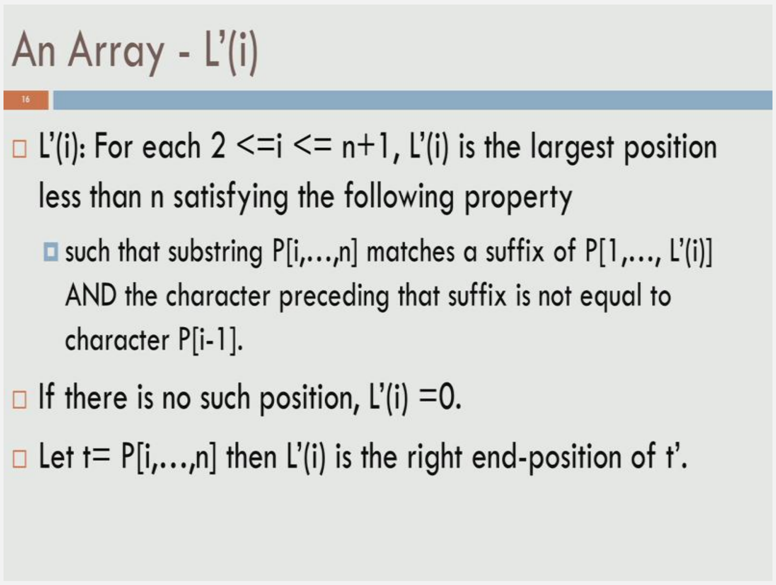
How to get the information needed for the good suffix rule? i.e., for a t, how do we find t`?

Here, it is left as an exercise, you can check that the good suffix rule is correct and you can do a proof by induction on the length of P. In other words, you should verify that if P occurs as a sub string of T, then the good suffix rule will definitely identify the first occurrence of P in T and the first occurrences from the left will definitely be discovered by this rule.

And the way to do this proof is by induction on the length of P, you can start off with a base case which is the case, when P is of length 1 and you can verify that the algorithm works and then make the induction hypothesis and verify it and you can actually use the weak induction hypothesis alone to complete the proof of this claim. This is left as an exercise to the interested student. Now, let us just step back and take a look at the two rules, the bad character rule focuses on the character mismatch, whereas, the strong character rule uses the sub strings and the matches which have happened.

Now, to implement the good character rule, to implement the good suffix rule how does I am get t prime for a given t. This is the implementation exercise and we are going to look a data structures to answer this particular question.

(Refer Slide Time: 26:58)



An Array - L'(i)

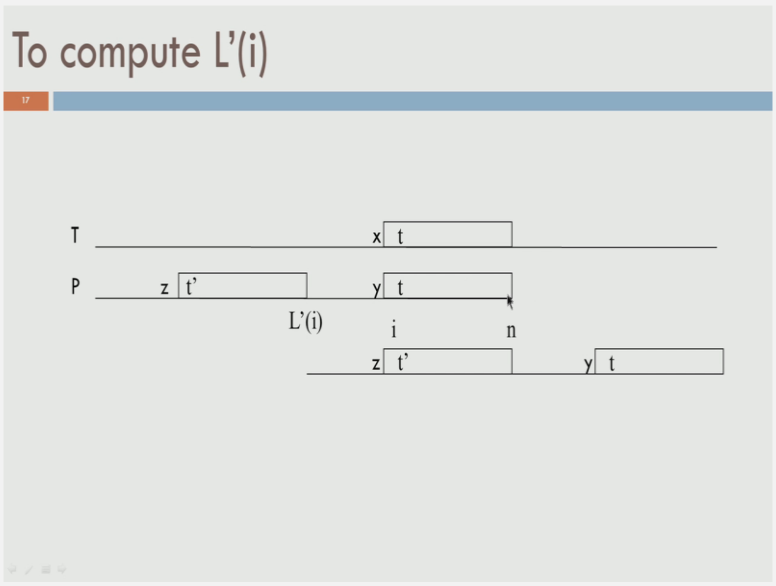
16

- L'(i): For each 2 <=i <= n+1, L'(i) is the largest position less than n satisfying the following property
  - such that substring P[i,…,n] matches a suffix of P[1,…, L'(i)] AND the character preceding that suffix is not equal to character P[i-1].
- If there is no such position, L'(i) =0.
- Let t= P[i,…,n] then L'(i) is the right end-position of t'.

So, one of the data structures is that we maintain an array which we call the l prime of i and let us just look at what l prime of i is. So, the l prime of i is a largest position smaller than the length of the pattern n which satisfies the following property. The property that is satisfied is that for the i, P i to n that is if you look at the suffix of the pattern P is starting at the index i, it must match a suffix of the pattern considered from the first character up to l prime of i, this is the value of l prime of i.

Further, the additional condition that must be satisfied is that the character preceding the suffix is not equal to P of i minus 1. So, P of i minus 1 is a character preceding P of i and that value P of i minus 1 must not be equal to the character preceding the suffix match in 1 to l prime of i and l prime of i is a largest value which satisfies this particular property. If no such position is found, then l prime of i is set to be 0 and we will refer to P of i to n, the suffix starting at i suffix of the pattern p starting at i with the letter T. Let us just visualize this array and it is role.

(Refer Slide Time: 28:41)

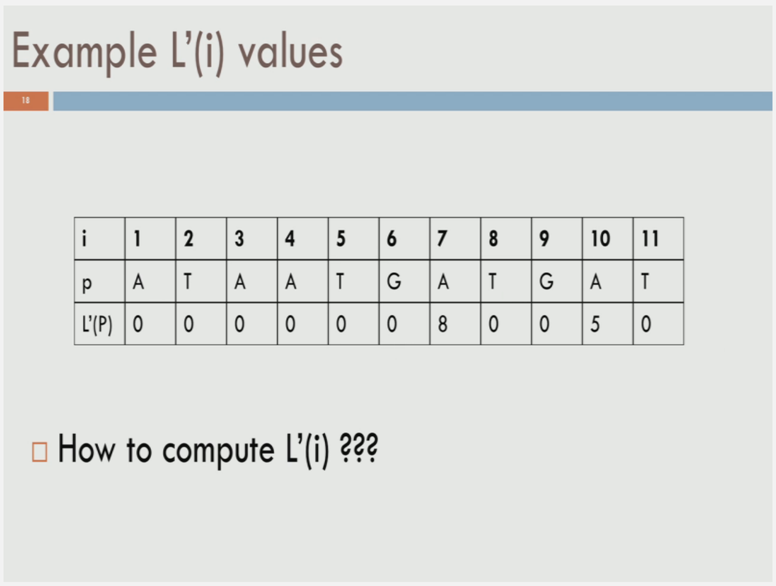


To compute L'(i)

So, let us just recall this is the alignment of P against t, this is a right end of P, small t is the matched portion and x and y is where there is a mismatch, we look at the largest value l prime of i such that the suffix and recall that i is the point of mismatch. So, i is the point to the right of the mismatch, in this case i as it is marked here. So, one looks at the suffix of the pattern string form i to here, which in this case matches with the target string t and l prime of i is a largest value at which this whole pattern t occurs as a suffix of P1 to l prime of i.

In other words, this is the position where t occurs just to the left. Of course, this could not be well defined, in which case we have the case two of the good suffix rule. So, observe that if you know l prime of i and if it is non zero, then that basically tells us how much we must shift be by, this is a very clear thing. So, one has to shift it. So, that t prime aligns with t and this l prime of i being the largest such value ensures that t prime was the copy of t and this is the first copy of t, when seen from the right hand side. So, therefore, l prime of i is a useful piece of information to compute.

(Refer Slide Time: 30:32)



Example L'(i) values

i	1	2	3	4	5	6	7	8	9	10	11
p	A	T	A	A	T	G	A	T	G	A	T
L'(P)	0	0	0	0	0	0	8	0	0	5	0

- How to compute L'(i) ???

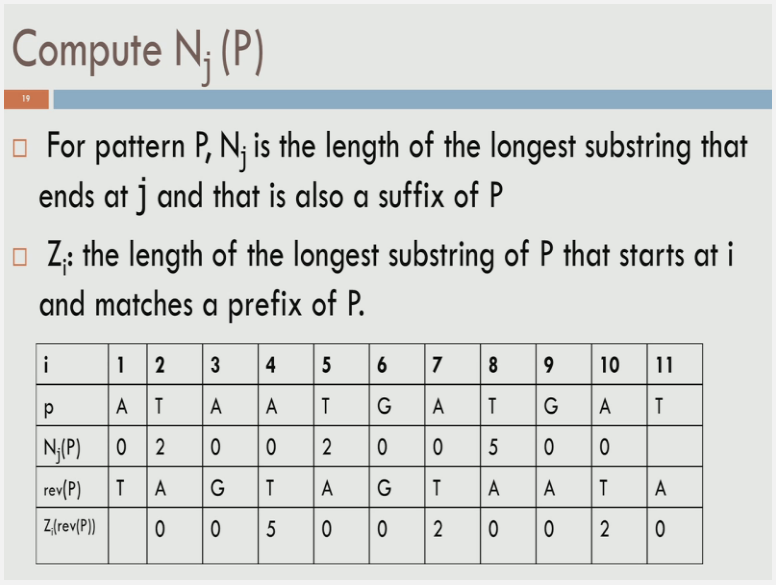
So, let us look at the example case for l prime of i for a pattern p. So, let us look at the pattern p. So, let us look at the l prime of 1, l prime of 1 essentially looks at the pattern starting from the first character up to the last character and ask for a preceding part of P that contains this whole thing as a suffix and obviously, that is not well defined. So, that is 0. Similarly, l prime of 2 is also 0, up to l prime of 6 which is definitely 0 and in all these cases that is because the string is too long to be contained as a suffix of a preceding portion.

So, therefore, it is a natural that all these values are 0. Let us look at P prime of, let us look at l prime of 7. In this case, the part of P that we are interested in a string starting at 7 and ending at 11 that is a string A T G A T. Let us see what is the prefix of P for which this is a suffix and for that match, G is not the previous element, G does not occur immediately to the left of this particular match. So, let us just see this. In particular, indeed we want let us look at the possible values for l prime of 7.

So, in particular let us see that l prime of 7 can definitely take the value 8, because the pattern is A T G A T and the value that occurs just before this A T G A T is A, on the other which is different from G. And therefore, in this case we can say that l prime of 7 takes the value 8 and you can also check that further to the right, we will not find a match of A T G A T in the strength P. Similarly, one can check that the largest place where A T occurs, such that G is not a preceding part is definitely the value 5, A T occurs here.

So, therefore, the l prime of 10, the candidate value is 5 and indeed it is 5, because just before A T it is A and not G. This is the additional condition that must be satisfied. Observe that A T also occurs here, but the symbol that occurs just before this A T is G which is same as the occurrence of G just before A T, therefore, l prime of 10 is the value 5 and this is 0. So, this did, this example is used to illustrate what the array l prime contains.

(Refer Slide Time: 33:56)



Compute N_j (P)

19

- For pattern P, N_j is the length of the longest substring that ends at j and that is also a suffix of P
- Z_i: the length of the longest substring of P that starts at i and matches a prefix of P.

i	1	2	3	4	5	6	7	8	9	10	11
p	A	T	A	A	T	G	A	T	G	A	T
N_j(P)	0	2	0	0	2	0	0	5	0	0	
rev(P)	T	A	G	T	A	G	T	A	A	T	A
Z_i(rev(P))		0	0	5	0	0	2	0	0	2	0

Of course, how does one compute l prime. Indeed, it is possible to compute with a lot of effort. So, an order of m n time, one can very easily compute l prime by doing a lot of pattern matches, but we will see that a clever way of maintaining additional data will help us compute l prime. So, we maintain now two arrays which we refer to the N array associate with a pattern P and indexed by j and also we look at another array that we maintain which is called the Z array which is indexed by i for the pattern P.

So, let us just look at the jth element of the N array. So, that jth element of the N array is the longest sub string that ends at the jth location which is also a suffix of P. Another way of visualizing this definition is to look at suffixes of P and see how long the suffix of P occurs or ends at j starting from the first location. So, that is N j is a length of the longest sub string that ends at j that is also a suffix of P, this is a formal definition. The visualization as I just said now is to look at suffixes of P and look at the longest suffix which ends at location j in the pattern P itself.

Kind of the mirror image of this particular array is Z array which is the length of the longer sub string of P that starts at i and it is a prefix of P. That is what we do is we look at the prefixes of P starting from the first location and look at the longest prefix which starts at i and the length of such prefix as it is maintained in the Z array and the length of the suffixes, the longer suffixes are maintained in the n array.

So, again let us do an example, let us look at the pattern string which is A T A A T G A T G A T. Now, let us just look at the value of the N array. Now, let us look at the longest suffix of the pattern which ends at the first location. So, let us look at the last symbol, it is T, whereas, the first symbol is A and therefore, the length of the longest sub string that ends at the first location which is also a suffix of P that length is 0.

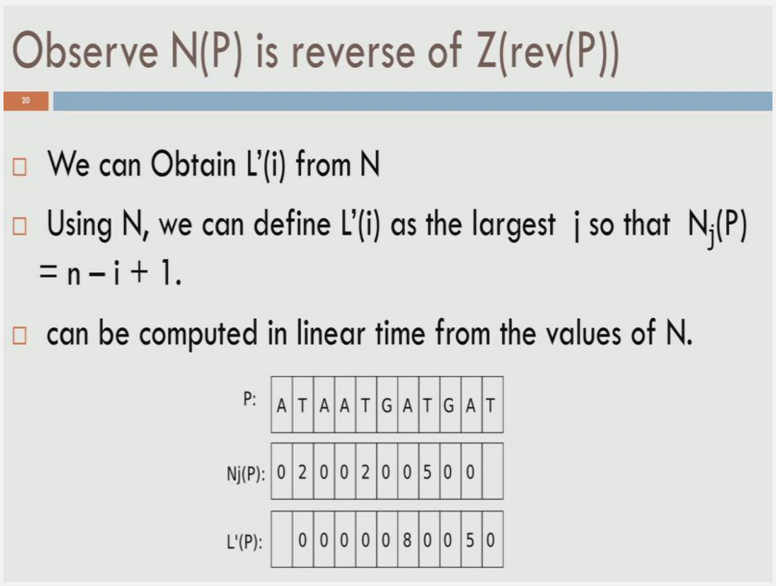
Let us look at the longest suffix, let us look at the longest string which is also a suffix of P that ends at location 2. Let us look at it, this is A T, this is a string and let us look at the last part, this is A T. Now, clearly this is a suffix of P and this definitely occurs in the locations P of 1 and P of 2. In other words, A T ends at location 2 in pattern P and therefore, the length of the longest string which ends at location 2 which is also a suffix is 2.

Therefore, using this definition you can now verify that these values are the appropriate lengths. We will also use this Z array along with a reverse of the pattern for a specific purpose and we will come to that in a short while, before that let us look at the reverse pattern P of the given pattern P, the reversal of the pattern P. In this case, the pattern is written backwards that is T A G T A G T A A T A. You can see that, that is a reverse of the pattern P.

And one can now see that the longest sub string of P that starts at i and matches a prefix of P is actually the reversal of the N array which is what you will see here. This is an easy exercise for the student to verify. So, therefore, we are. So, for maintained three arrays. So, one is a capital l prime, the second one is the N array and the Z array. And the goal is to compute the values of the entries in the capital L prime array and for that we are going to use the N array and the Z array.

And recall that I have just mention that the Z array will always be relevant to the reversal of P, in other words it will turn out that the reversal of N of P is exactly the Z of the reversal of P which is what we have seen in the previous example.

(Refer Slide Time: 39:22)



Observe N(P) is reverse of Z(rev(P))

20

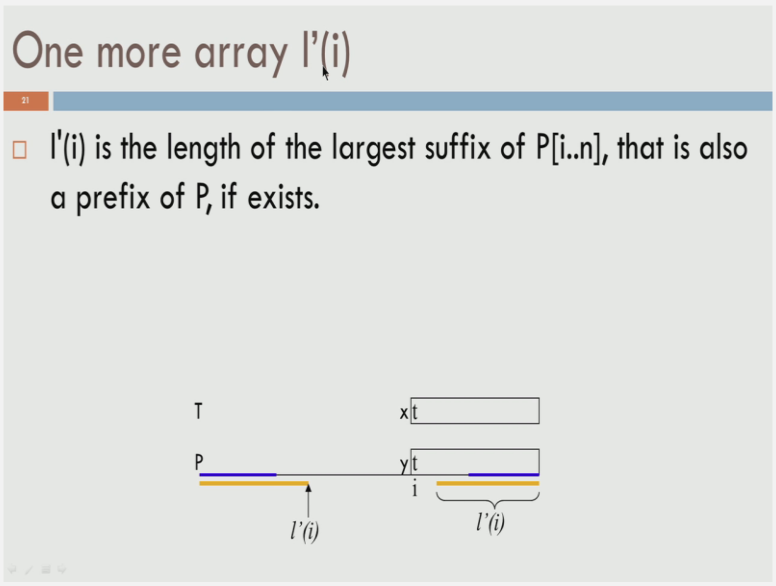
- We can Obtain L'(i) from N
- Using N, we can define L'(i) as the largest j so that $N_j(P) = n - i + 1$.
- can be computed in linear time from the values of N.

P:	A	T	A	A	T	G	A	T	G	A	T
Nj(P):	0	2	0	0	2	0	0	5	0	0	
L'(P):		0	0	0	0	0	8	0	0	5	0

So, now we present the rules for obtaining the value of L prime, again we present these rules without proofs, but the examples support what we actually do. So, to obtain L prime of n, L prime of i that is for a particular location i, the value of L prime as defined a few slides back is taken to be the largest j satisfying the property that N j of P is n minus i plus 1. So, here observe that for L prime of i, we use the array N and we define L prime of i to be the largest value of j, such that N j that is the jth location in N array has the value which is n minus i plus 1.

Now, once we have the N array, it is clear that L prime can be computed in linear time by performing a single pass over the N array. We need one more piece of information which would be used, when the first case of case two or the first possibility of case two does not apply.

(Refer Slide Time: 40:56)



One more array l'(i)

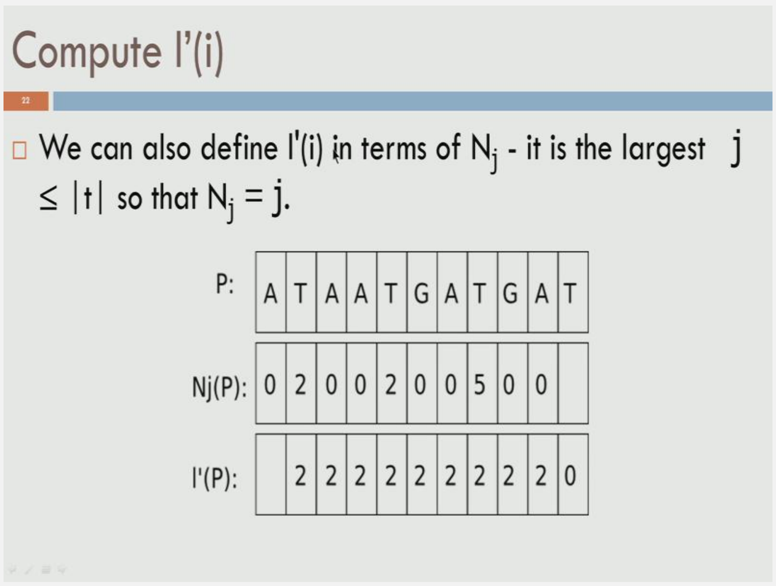
- l'(i) is the length of the largest suffix of P[i..n], that is also a prefix of P, if exists.

T
x
t
P
y
t
i
l'(i)
l'(i)

For this, we introduce a fourth array which we referred to as l prime. So, let us just look at l prime, again the l prime is associated with the ith location in the bottom. L prime of i is a length of the largest suffix of the pattern starting at the location i and ending at the location n and this was also be a prefix of P, if it exists. So, let us just look at this picture. So, this is the use case, this is a part which is matched and there is a mismatch here.

This is what we have been discussing. So, for. Now, l prime of i is a suffix of this particular pattern which is P of i to n and l prime of i is the length of this portion which also satisfies the property that it is the prefix of the string. Remember that this could not be, it could be the case of, this is not well defined. But, if it is well defined, l prime of i takes this particular value, otherwise it takes the value 0, because it is a length. Because, if this property is not satisfied, l prime of i will take the value 0.

(Refer Slide Time: 42:06)



Compute l'(i)

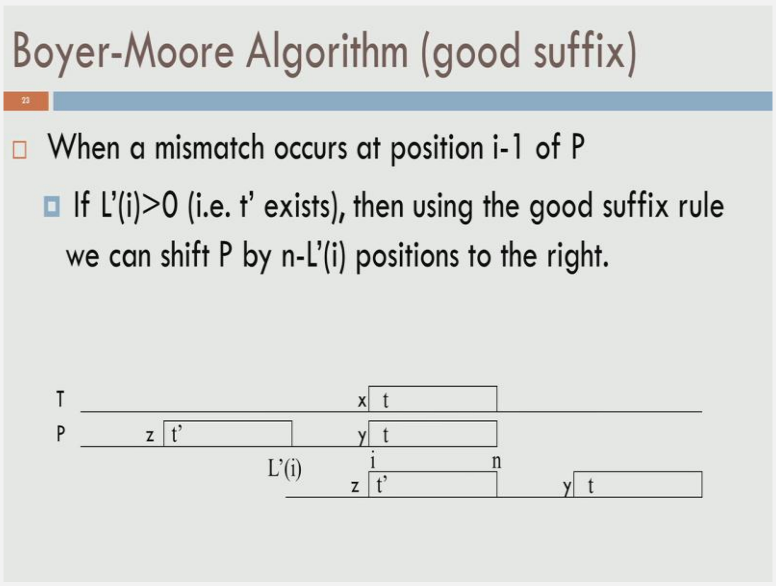
- We can also define l'(i) in terms of N_j - it is the largest j ≤ |t| so that N_j = j.

P:	A	T	A	A	T	G	A	T	G	A	T
Nj(P):	0	2	0	0	2	0	0	5	0	0	
l'(P):		2	2	2	2	2	2	2	2	2	0

How does I compute l prime of i, the value l prime of i can be computed again from N j and in this case, we will use the reversal of N j and this can be checked. So, we save the l prime of i is the largest j which is smaller than t, such that N j takes the value equal to j. Let us just recall that t is a shortcut for this suffix P of i to n that is where i comes into this picture.

Therefore, we interested in l prime of i, you are looking at the suffix which consists of the letters in the pattern, starting at location i upto location n, I refer to this as t, this has a certain link we look at the largest value j which is smaller than or equal to t, such that the value N j in the N array is exactly equal to j. Again, this is an illustration of this and this can be worked by an interested student by hand.

(Refer Slide Time: 43:11)



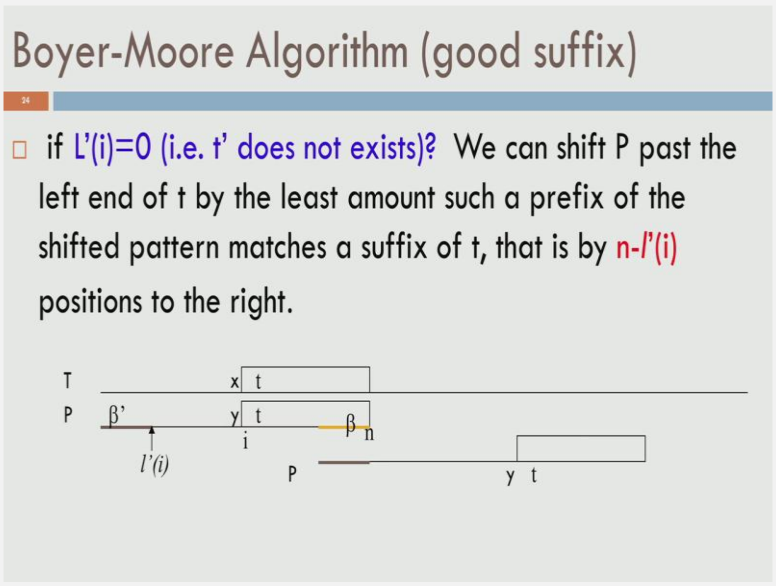
Boyer-Moore Algorithm (good suffix)

- When a mismatch occurs at position i-1 of P
  - If L'(i)>0 (i.e. t' exists), then using the good suffix rule we can shift P by n-L'(i) positions to the right.

This brings us to the Boyer Moore algorithm. So, now what we will do is to illustrate the steps of the algorithm using the arrays that we have set up. So, one can just concentrate on just a steps in the algorithm and here this is the Boyer Moore algorithm using the good suffix rule. So, when a mismatch occurs at position i minus 1 of P, let us just see this t is the matching pattern, P is aligned with t, this is the right to left comparison. The mismatch occurs at location i minus 1 of P and it occurs with, let us say this index is k.

The first step is to check if this pattern t occurs to the left as a pattern t prime. This is obtained by looking at the l prime array, if this value is greater than 0, then it says that t prime does exist. What we do is we shift n, we shift the pattern P by n minus l prime of i positions to the right. So, let us just see this, this is l prime of i. So, what we do is we push the whole pattern starting from here, n minus l prime of i positions to the right, in which case the alignment is that this t prime gets aligned with this t. So, this is the shift.

(Refer Slide Time: 44:35)



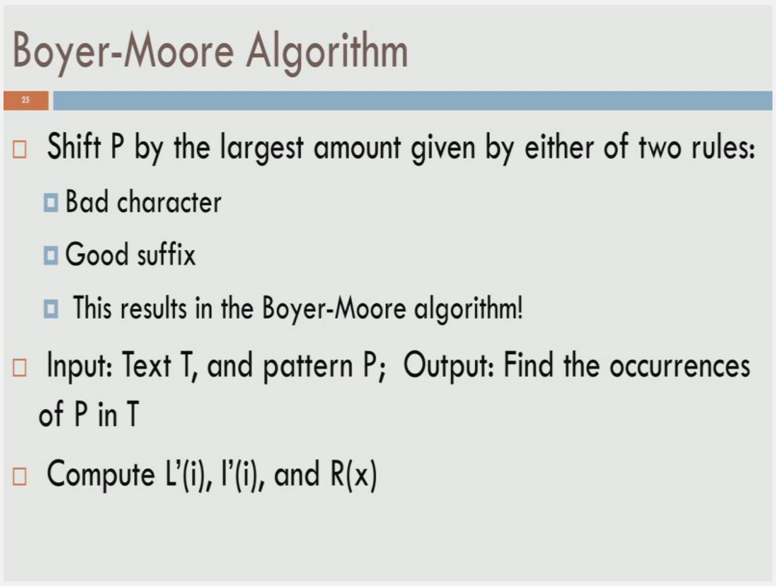
Boyer-Moore Algorithm (good suffix)

- if L'(i)=0 (i.e. t' does not exists)? We can shift P past the left end of t by the least amount such a prefix of the shifted pattern matches a suffix of t, that is by n-l'(i) positions to the right.

Now, we have to worry about the case where l prime of i is equal to 0. In this case, it means that the pattern t prime does not exists, in which case this means the t does not occur to the left of it is current occurrence. Then, it says that we can shift P pass the left end. So, that, that is the pattern P is shifted. So, that a prefix of P matches the suffix of t and the way this is done is to compute the value l prime of i and shift the pattern P to the right by n minus l prime of I and this is the shift that is recommended.

By the shift, the pattern t goes for that to the right and the portion of P which matches with a suffix of t is what will match after the shift, this is the good suffix rule.

(Refer Slide Time: 45:32)



Boyer-Moore Algorithm

- Shift P by the largest amount given by either of two rules:
  - Bad character
  - Good suffix
  - This results in the Boyer-Moore algorithm!
- Input: Text T, and pattern P; Output: Find the occurrences of P in T
- Compute L'(i), l'(i), and R(x)

Now, let us just complete the whole algorithm, what we do is we select a shift value for P which is a largest one given by either of the two rules. That is we take the value, shift recommended by the bad character rule and we look at the shift given by the good suffix rule. Recall that some matches have happened which is used by the good suffix rule, there is a mismatch that is happened, that is taken by the bad character rule and each of these two rules specify a shift value and the Boyer Moore algorithm takes the larger of the two shifts which are recommended by these two rules and that is what is used to shift the pattern P to the right and this is what the whole Boyer Moore algorithm specifies.

We have not given the proofs of correctness of these rules. What we have done is set up the whole data structure and given examples in the slides and we have enable a potential implementation by an interested student. So, let us just summaries, the input consists of a text T, a pattern P. The output is the occurrences of pattern P or a response that pattern P is not found in T. The algorithm is very simple, we have to look at the description of these three arrays which is L prime, I prime and R.

Recall that R is used by the bad character rule, the capital L prime and the small l prime are used by the good suffix rule. There is also an additional array which is called the… There are also two additional arrays which are called the N array and the Z array which are used for the values, for the computation of the values in the small l prime. So, this completes the presentation of the Boyer Moore algorithm.

Thank you for your attention.