Lecture Transcript
String Matching Algorithm

Hello and welcome to this next lecture on Data Structures and Algorithms. Starting today, we'll discuss an interesting problem call string matching and the problem on string matchingis about finding a textual pattern, let's take P in a set of strings s1, s2 and so on. You come across this very often in textual editors such as Vim or Gedit or you have this Unix command Grep. This pattern P could be any regular expression. So, we'll discuss both extremely Naive algorithms as well as efficient algorithms for such pattern matching. So, formally the problem is to find all the valid shiftswith which a given pattern P occurs in a given text T. More generally, the text T could be a set of strings as we pointed out in the previous slide. And here's an example, you are provided the string s and below is the pattern P, so you want to find instances of S T Y in the string s. You start from the left most end s = 0. You find that there is a mismatch at the third position though there is a match in the first two positions. You can then think of moving the pattern to the right, scanning at the second character position of the string, you find as mismatch right away. Further down you again find a mismatch at E and then at P. However, at position five which corresponds to index four of the string. You find a match at all the three positions S T and Y.

So this basically results in one pattern match. You can continue this process if you are interested in finding all the pivot and matches. You might stop here if you are interested only in finding a single pattern match. So, assuming that interested in all pattern matching you look at index five, find a mismatch right away, index six there is a mismatch. Now, do you need to proceed? Well, doesn't make sense because you certainly cannot let the index overflow n - m or the index position corresponding to the number of positions in the string minus the number of positions in the pattern. So, this is the Naive string matching algorithm that we just illustrated given a text T of size n and pattern P of size m. You keep track of the shift index s at every point of time. So the shift index s is what you preserve into the text T. The pattern P at this instance of time will be scanned starting from this position s. So for s, ranging from 0 to n-m, and we are interested in matching the entire pattern. So s will, we restricted to number of positions in the text T minus the number of positions in the pattern P. So, for this range of s, we are going to keep track of the position j.

The position j is basically the index into P. So, you vary j over the positions in P and do it till j is less than m. So j is start form the 0 and maximum value of m - 1. We are going to match T[s + j] with P[j] at every point of time, and in case there is a mismatch you terminate. So, continue scanning P as long as there is no mismatch with T. So once you are at the end of P, you flag a match print 'Valid at shift s' that what's it means to have a match and get interested in all matches and because of that we continue the scan. If we are not interested in all the matches, we could break right after printing the valid statement. Analysis of this Naive algorithm. The inner loop is going to be called at max m times order m times.

This is assuming that you have a match at every position. So the worst case situation is as follows. T is aaaaaa and P is say aa, m is 2 and n is 6. So you are going to scan all the positions of P for every position in T because match occurs all n - m + 1 positions. So this will result in c1 times in the worst case for each position s, and overall you will have to do this for every position in T which is n - m + m.

So overall there are (n - m + 1) into c2 calls overall and this results in (n - m + 1) times m overall complexity. What will see in some of the subsequent discussions are more intelligent algorithms which avoid this brute force computation by explicitly caching and memorizing. Some of the computations that we've already done and reusing them. Some of the algorithms such as dynamic programming are in some sense brute force because they do consider all possible configurations but they leverage previous computations in order to avoid redundant or repetitive computations. We will see some of that flavour in the algorithms that follow.

Thank you.