Lecture Transcript
Finite Automaton Algorithm

Hi and welcome to this next lecture on String Matching. Recall our Naive algorithm for string matching which was basically looking for all matches of the pattern P in the text T starting at every position in the text. What we were concerned about with the Naive algorithm was that there are potentially prefixes of the pattern P which also occur as suffixes in the text that was already matched. So, imagine that we ran a pattern matching P on the segment and we realize that there is a mismatch. How will the process, the some segments of P that already matched T would we not want to leverage them and that's precisely what we'll discuss today through the construction of a finite state automaton. What is a finite state automaton? It's a simple machine for string matching. A finite state automaton has an associated alphabet and given any observation from the alphabet the automaton makes transitions from something called states, from one state to another state.

So, this is how a finite state automaton looks like. The state S1, state S2 on seeing a symbol a1 from an alphabet sigma, you might make a transition to S2. However, on seeing another symbol a2 from sigma you might make a transition to S3. You can also have self-loops on certain characters. You cannot have outgoing transitions to two different states on the same symbol, so it's not possible to have S1 go to S3 on a1. Interested in deterministic automaton, a transition of this sort is allowed in non-deterministic automaton. You also have something called an accept state. So, on a particular symbol a3, I might go from S2 to S4 and call S1 an accept state. What does this automaton do? It basically accepts strings of the form a1 a3. Of course, we haven't completed the automaton, so on S2 if I see a2 I might go to S3 and keep looping around as I see a1. Unfortunately, I haven't created an exit back from S3, so imagine that I have a transition from S3 to S1 on a2 as well as a3. For S2, I might loop back on a1.

So, now we can complete the set of strings that this automaton will accept. It accepts, a1 a3 and I am assuming here that I am beginning with state S1. It also accepts a1 a1 a3 and a1 a1 a1 a3. So, it basically can accept a1 followed by any non-zero numbers of a1 followed by a3. It can also accept strings prefixed with a2 a1 a2 or a3 followed by a1 a3 and so on. So, it's a whole bunch of strings that it can accept. Our goal is however to construct a finite state automaton for a fixed pattern P in order to find all occurrences of the pattern P in the input text T. Now, the pattern P will be assumed to be fixed whereas the input text can vary. Now, we will invest more time in constructing the pattern P and what will, there benefit will enjoy is that we will be able to quickly process and scan the text T for instances of pattern P. So formally a finite state automaton M is a 5-tuple which consists of finite number of states, Q a start state q0, and accepting state A, so recall that we denoted accepting state by two circles, concentric circles.

The start state was denoted by an incoming arrow. sigma is a finite input all alphabet, on the previous slide sigma was a1 a2 a3. delta is a transition function which takes you from one state to the other, so delta on S1, at S1 with input a1 took you to S2 and so on. So you can list all the deltas associated with the finite state automaton illustrated on the previous slide. Now what you want is to construct an automaton that exactly accepts strings corresponding to the pattern. So, let's illustrate with an example. We have pattern P = abc and we need to see what are the matches of abc in to the text abababc. I have constructed this finite state automaton corresponding to the pattern. It has 3 states, states are 0, 1, 2 and 3, so these are accept state, start state and two intermediate states. The sigma consist of a, b and c, which is alphabet. The delta as specified in this finite state automaton is as follows, delta on 0 with input a takes you to state 1, delta on state 1 with input b takes you to state 2, delta on state 2 with input a takes you back to state 1 and so on.

Now, does this automaton accept only instances of this pattern P? You can convince yourself that for any sequence which has not abc. For example, cba. You will be taken back to the start state 0. So, on c there is nothing you can do here, on b you keep looping back here, on a you move to 1, from where you move to 2 on b, and from 2 you move back to 1 on a. So, if I had cbabc, I will start with 0, not move away from 0 until I see this a, and then b and then c will be in the accept state 3. However, if I see an a instead here, which is ababc. So when I see this b or rather by the time I meet b, I will be on state number 2. However, when I see a I will be back to state number 1. Which means I've already registered the first a that I should have matched against this pattern. The rest is just going to be routine just b followed by c. How do you construct such a pattern that accepts matches only corresponding to abc and doesn't accept a match corresponding to any other character sequence while preserving some memory about the characters that I've already been matched so far.

Now, we will illustrate the algorithm for constructing the finite state automaton just slightly more difficult example. The pattern is ababc. And as I have pointed out, we need to keep track of character sequences in P that I've already been matched in the text T. So goal, keep track of character sequences, the character sequence and by sequence, I actually mean a prefix, character prefix of P that has already been matched so far. So, how do we do this? Well, we going to first of all assume, the sigma is exactly what we see in the pattern a, b, c, we will also assume the set of states q, to correspond to as many positions as there are in this pattern, so we'll have position 0, 1, 2, 3, 4, we'll also have a position 5 corresponding to the accept state. Now, before I write down my algorithm, let me specify what I need to keep track of this character prefix of P, that has already been matched so far. So, given any input sequence (x), I am going to define a function sigma that takes as input x and outputs length of longest prefix of P that is a suffix of x.

Now, what is this x? Well this x is what has been matched so far already in the automaton. x is a sequence of characters already matched in automaton. So, I would like to look at where to hop back into the automaton, so that the prefix of P that is already matched as per the suffix of x is registered or memorised. Let us just complete this definition, so this more formally is nothing but the max over all indices k. Such that, the pattern Pk which is the first k characters of P prefix of P happens to be a suffix of x. So, we're going to denote this as the subset operation but what we mean here is suffix of. Now, let's try and see what this would mean for our sample pattern a b a b c? So, I am going to start with my states 0, 1, 2, 3, 4 and 5, 5 being the accept, 0 being the input, and I need to now identify where my arrow should be going. So on state 0 when I see an a, I go to 1 from 1 I go to 2 when I see a b, on 2 when I see another a I should go to 3, from 3 when I see a b I should go to 4, from 4 when I see a c go to 5.

This is routine, so suppose I have so far seen the string a. Next when I see a b I know that I am going to move to state 2. How about c? So, when x is a c, I am going to ask this question? what is the length of longest prefix of P? That is the suffix of x. So, when I suffix of x, it must include the last character c. An unfortunately, that's not possible you have to move back to 0. So when I see a 'c' I am going to move back to zero. What if I see an 'a' instead of 'c', 'aa'. What is the length of the longest prefix of 'p' that is the suffix of a well it just one. So, on seeing an another 'a' I am going loop back. So please note, we are already provisioning for keeping track of character prefix of 'p' that is already been matched so far. So, let's continue our journey. What if I see a 'b' next, well I know if I'll see a 'b' next I am moving on to state number two. And there, since there are no other options I'll consider 'ab'.

Now, suppose I add an 'a' to this, string 'ab' is what I have when I have reached state number two. And If I add another 'a' I am on the state number three if I see a 'b' next what do I do? Well, there is no transition possible. What is the length of the longest prefix of 'p' that is the suffix of 'x'? Well, it turns out that it is zero. So when I see a 'b' in state number two I have no option for to go back to zero. This is also intuitive for the string 'abb' you basically have to start any subsequent matches from scratch from the start of 'p'. Now, what if I see character 'c' next. What if that's the next character I see 'abc'. Well, the length of the longest prefix of 'p' that is the suffix of 'x' is still zero. So I am going to mark this arrow for two transitions 'b' and 'c' from two. Now the only other option now is 'a' and we know that if 'a' is the next character I have to move on to state number three. And at state number three again I have the option of saying a 'b' which takes may to state number four. If I another 'a' what do I do? So note I am at state number three and I've seen an 'a'. What is the length of longest prefix of 'p' that is a suffix of 'x' and it just turns out to be one which means I'll move from state three to state one when I see an 'a'.

So here again we are registering a character that is already been seen. This could be handy. What if my next character was a 'b'. Well we already know, we go to 4 and now what if my next character was a 'c'. What is the length of the longest prefix of 'p' that's the suffix of 'x'. Unfortunately, it is zero so this will mean that at state number three when I see a 'c', I go back to zero. We move on to state number 4 to complete the story now. What is the next possible character I see, a 'c' will get been accept and 'a' the length of the longest prefix of 'p' that is the suffix of this new string is basically one so again this will take me back to state number one. A 'b' will take me back to state number zero and a 'c' will take me to the accept state. So this is the story of this longest prefix 'p' that is the suffix of 'x'. Now what does it translate into? Summarily, what we have saying here is that the delta, the transition from a particular state 'q' on seeing any particular character alpha will be sigma with the prefix on the pattern 'p' that led me to 'q' that is 'pq'. But appended with alpha this exactly what we see here. So when you are at state 3 the q is 3, the Pq is a ba.

Now corresponding to different values of alpha you've different transitions. So an alpha of 'a' takes you to sigma a baa for which the corresponding sigma value is 1. So you can see that delta qa equal sigma Pq alpha should give you the destination state for almost every q. You can continue this process and convince yourself that indeed these definitions suffices for us. Now how do you translate this observation into an algorithm? The intuition behind the algorithm is as follows. So what is the desired invariant of this algorithm? When I have seen the first i characters starting from position i of T. I would like to be in a state which corresponds to the length of the longest prefix of p that is a suffix of the pattern Pq which has taken me to that particular state. This should be the state after I see 'i' this is the basic idea after seeing the first i positions and here's the algorithm that will help you maintain this invariant. So, let's call this algorithm construct finite automaton for a given pattern p with sigma. We'll assume that the pattern p has length m and we know that our states will be ranging from 0 to m.

So we are going to start from state 0 and go until m. What we did in our informal yet illustrative process here was we iterated over each character alpha in sigma. The next thing we did was keeping track of suffix is sub x and trying to find the longest prefix of P. So this will mean that 'i' will keep scanning backwards from wherever I am and what it means to be wherever I am is basically Pq with the next character. So let's make that explicit here, scan backwards from the string Pq suffix with an alpha to find state to go to, and what is the state? It's basically sigma of Pq alpha, but how do you get this sigma? Let's fill that up. So we are going to keep track of the position and scan backwards using a variable k. So k is going to be set to be the minimum of either the last position m and q + 1, this is because we are trying to look ahead. Ahead of the current state and see what this alpha should lead me too. So as we can see here, when q equalled 3 and when alpha equalled b our desired destination was 4 and there for q + 1 is important. So k is a min of (m and q + 1) of course if q + 1 itself as over flown, we can only do with m.

So let's see what to do with k? So we are going to say while Pk the prefix ending at k is not a substring or doesn't match the prefix of P(sub)q alpha, we are just going to do k = k - 1 and once we find that P(sub)q is indeed a substring of P(sub)q alpha and this will of course with the case when k become zero we then going to set delta of (q,alpha) to be k. Of course, we need to complete the for loops. This has helped us complete the delta for every state and for every alpha and basically delta (q,alpha) points to the length of the longest prefix of p, that is a suffix of P(sub)q alpha. So, what we will do now is having constructed a finite state automaton for a,b,c will try and match it to the text T and we'll actually see how we are able to leverage the memory that the finite state automaton gives us. The state transitions have been also tabulated below here, so this is nothing but our delta, which was obtained by virtue of our algorithm. And now we can see the algorithm in action. So starting at position 1 and you see input 'a' the delta for zero tells you to go to 1, that's what we've done. When you see a b you go to 2, however, when you see and a you go back to 1. So a, b the next a takes you back to 1.

The next b takes you back to 2 and then you do another 'a' for which you go back to 1, a b which takes you back to 2 and now finally you are at state number 3 when you see a c. So a match was just a,b,c and we avoided going back to zero all the way when we saw this a onto occasions. So we did save at least 2 computation or matches which a brute force algorithm would have been incurred. So here is a simple finite state automaton algorithm specifically for matching an input text T. So given the delta the m and the T, if s is a shift index to T you start with the start state q=0 and for i=1 to n as you scan position i of the text T. You make a transition from q to delta of (q,T[i]), if you have reached the final state then you accept and you also print out the position on match. Of course, you can continue this whole process what we didn't do in our construction was complete a transition from the accept state, but please note that these transitions will also exist at the accept state. As far as complexity is concerned this finite state automaton algorithm was simply a linear scan, just scanning every position in T once. So as you note this is been a simple linear scan and therefore, the order of complexity is just order n. However, there is a hidden cost and the hidden cost is computing the delta for the pattern P.

The delta for P is offline and therefore, doesn't have to be computed for every specified input text. So, this is a onetime cost and it makes sense to incur significantly larger one-time cost if you want to be effective for a large number of input text. Let's however, note what the complexity of computing delta is? So note that in this algorithm for constructing finite-state automaton, we had once scan over the pattern P, the outer loop. Now for each element of the alphabet sigma, I am going to incur the cost of running the loop on m. So you basically you need to multiply m with the size of sigma. Is that all? Well, I'll need to go, keeping going back on the k and in the worst case I might have to go back all the way to the left. So the third aspect is this while, which will again incurred and a factor of m his multiplied and finally even to check whether Pk is prefix that matches to the suffix of Pq alpha. I might incur in the worst case and order m operation. So again multiplying this with m, the overall complexity is m cube sigma.

Thank you.