Lecture Transcript
Knuth-Morris-Pratt Algorithm

Hello and welcome to this next lecture on Data Structures and Algorithms. Today, we will conclude our discussion on matching patterns on strings with the very well known Knuth Morris Pratt Algorithm also abbreviated as the KMP algorithm. Recall our discussion of string matching on a pattern using a finite state automaton built from the pattern. We devise an interesting look up table for the transition function delta of the finite state automaton with the goal that we are able to find all matches of the pattern on the input text in time that is linear in the number of characters in the text.

Now, we are going to leverage that same idea but instead of constructing the entire finite state automaton, we will basically make use of the delta kind of look up constructed on the pattern to directly match the text while avoiding as many backward traversals as possible in the text. In fact, we will show that the number of backward traversals on the text will be upper bounded by the length of the pattern itself. In fact, the same idea will be used to construct the lookup table for the pattern itself. So, here we go. The naive algorithm had no pre-processing on the pattern but required a significant amount of time for matching the pattern on the text. So, order of (n-m+1) into m. The Rabin-Karp algorithm incurred linear time in pre-processing the text. Recall that this was using hash functions, but the matching time did not reduced significantly. In fact, it was still order of (n-m+1) times m.

The finite automaton gave you a linear order n matching time, but the construction of the automaton turn out to be expensive order m cube times size of the vocabulary. We are going to leverage the construction of the finite automaton, kind of getting rid of what was unnecessary in the finite automaton, which is remove unnecessary baggage and leverage the delta function to reduce a preprocessing time to order m while still maintaining order n matching time. So this is the story. So the idea of linear time string matching algorithm using KNP avoids computing the transition function explicitly, but instead, leverage is the idea by which delta was computed to find the so called prefix function pi. And, this prefix function pi links to the index of the pattern P that has the longest prefix for any index i of the pattern i.

So, Pi[q] is maximum value of the index k, such that Pk, the string from 1 to k in P happens to be a proper suffix of the string Pq. So, k is therefore, be the length of the longest prefix of P that is a proper suffix of Pq. So, recall that this was exactly the idea that was used to compute delta. We're going to levarage Pi[q] directly. So, a brief recap if your pattern P is a b a b a b. We initialize P[i] 1 that is the index k into the longest prefix of P that is the proper suffix of P[i] that is a, so there is actually only one character of proper suffix would only correspond to the empty prefix. So P[i], P[i] 1 is zero, what about Pi[2]? We have actually listed proper prefixes of Pi[2] and proper suffix as well it is just b, we find that there is no prefix of P2 that is a proper suffix of P2, so Pi[2] is, therefore, zero. At the next level, we find that it is indeed possible to find a prefix of P3 that is a which is a proper suffix of a, ba. So Pi[i] is therefore 1 pointing to a. How about P4, well there is indeed a prefix ab that is a proper suffix of ab, ab. So this gives you an index of 2.

Now we'll see how to compute this 2 without having to enumerate the proper prefixes and proper suffixes at every step. In fact, this is exactly the idea that is going to be used for string matching itself. We are going to determine the number of characters that need to be skipped in the pattern mining process. These are characters which we are assured of not having matches and we will basically make use of the lower values of Pi in this computation of Pi for higher values of q. So we continue this and we find that for Pi[5], the length of the longest prefix ab, a which is a proper suffix of P[i] 5 is 3. Pi[6] has a longest prefix of length 4 which is a proper suffix of P6, so given the table let's try and do pattern matching on text T, we'll revisit the construction of that table in a more efficient manner once we have learned sufficiently from the KMP pattern matching process itself. So the idea is we compare each character of P with text T to obtain partial matching pairs and you keep track of the number of characters that are currently processed or matched for pattern P and text T.

So, NCM stands for Number of Characters Matched. So Pi and NCM is going to tell you the longest prefix of P that is a proper suffix of the pattern that ends at NCM. So this denotes the number of characters that need to be skipped in text T. So, NCM - Pi [NCM] are the characters that you know will certainly not match in text T, which you could skip. Let's illustrate this, so we started a and c, there is no matching, we can proceed we are only interested in looking up Pi when there are some matches when there are some partial matches. So, yes that the starting at the next character we do find some partial match until we hit c and a which don't match. So we, in all we have 4 positions. What we are going to do now is, decide if we need to skip and how much we need to skip? So we look at Pi[4] which corresponds to the length of the longest prefix that is a proper suffix of the pattern P 1 to 4 ending at b, and it turns out that this is ab. So, we know that this is the length that we should be careful about. So we start matching characters at the next position and we find that we have matches until position 4. Position 5 of the pattern there is a mismatch.

Now, how much should be move to the right? We look at pi[4] and we find the length of the longest prefix of the pattern that happens to be a proper suffix of the text and that is two. So what we know is we can move to the right and be able to start matching at the next position following ab. So we need to start matching at this a matching with this c. So we know that there are chances of continuing this match starting at the next position because we have this prefix match ab with this match ab. So, at best what we can do is move two positions to the right which is basically 4 - pi[4]. So pi[4] matters to us, the remaining don't matter to us, so we subtract pi[4] from 4 and move two positions to the right. We know ab match ab and we actually need to see if the next character here a will match c. And we unfortunately don't get a match between a and c. Well, what can we do? Can we still solve something? Well, we know that two positions are already matched.

So what we can do therefore is look at pi[2]. Now, what is pi[2]? What is the length of the longest prefix of ab which is a proper suffix of ab, it is basically zero, we know from this table. So since there is actually no match, no prefix match at all. We look at 2 - pi[0], which is 2. So we can confidently skip these many characters, and start the matching at the next position. So we start with matching a and c, nomatch, we proceed. Well, starting at the next position we have three matches and again what do we find, there is a mismatch at the 4th position we look at pi[3]. The length of the longest prefix of aba that happens to be a proper suffix of aba is 1 and that's basically a here, and I know that this a will match this a. So therefore what I can do is skip the characters that come in middle. And, what does that mean. It means skipping 3 - pi[3] which is 1 and this is two characters, so skip these two characters.

So, you can see that a lot of unnecessary matching has been avoided simply by making use of the pi. We continue a match it a, a didn't match band now there is actually no scope for further matching because we have actually exceeded, T, the right threshold of T. So we need to do this only for n - m steps on the text T. So here is the overall KMP algorithm which assumes that we have obtained the pi table already. So, you compute the prefix pi we'll soon see that the same process can get you pi, the same process of matching the pattern P on the pattern P will get you pi. But for the time being we'll assume pi is already known. Now until position n - m, here what you do, you keep checking if P[q + 1] is not equal to T[i]. So,while q is greater than 0 and P[q +1] is not greater than T[i], you set q to pi[q]. So, you basically are looking back and seeing well where do I need to reset and then check if P[q + 1] = T[i], if yes then increment the value of q, if q lands up being m the end of pattern P you have actually got a shift at i - m wherethere is pattern match and this you set q to pi[q].

So, the skip of NCM - pi[NCM] is actually happening here. Now, how do you compute the prefix pi? How about running the same process on the prefix P replace the text T with P. Start with the left-most index of P and construct higher values of pi based on matches or mismatches at the lower level. So, here is the compute prefix function. So what you note is that we are maintaining an index i into P but treating P asT, and we again have this skip here where you set q = pi[q] but note that we haveset pi[1] to 0 which we've already discussed. We are not incrementing q anywhere here. We are only looking up q for q = 0 initially. Once P[q + 1] = P[pi] once you have actually visited this then q is set to q + 1 which means you are looking for the next look up and pi[i] is set to q, the length of the longest prefix of P[i] which is a proper suffix of P[i] happens to be Pq. What about running time?

So, first of all we are iterating over the pattern P m times. There is of course a bid of bad tracking that happens when you set q to pi[q]. One can show that this actually cannot happen more than m times. So the overall computation will be order of m, likewise, the pattern matching of P on T happens n-m times overall over the index i. However, there is this additional backtracking, this might create a problem. However, one can show that the number of backtracking is good, upper bounded by the length of the pattern itself. For the details you can read the proof in CLR the section on Knuth-Morris-Pratt Algorithm. So the overall run time is basically order n.

Thank you.