Second Level Algorithms

Prof. Palash Dey

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

Week – 11

Lecture 53

Welcome to the 53rd lecture of the second-level algorithm course. In this lecture, we will start another very important topic, which is string matching. So, let us begin. So, the input consists of two strings: the text string T[1..n] and the pattern string of length m, and the output is does the pattern string appear somewhere in the text? In particular, does there exist an s such that T[s+1..s+m] matches with the pattern string P of length m. Hence, s must be between 0 and n minus m plus 1 or n minus m minus 1, okay. So, this s is called a shift. Whenever we use Ctrl+F to search for something in a long text or in an article, then this algorithm or this problem becomes very useful. Let us first see what the naive algorithm is. For any problem, it is often wise to begin with the naive algorithm and then try to improve its performance. So, what is the naive algorithm? So, we are given. So, let us call this function string_matcher. The text is the input, and the pattern.

So, these are the inputs. And we simply iterate over all possible values of the shift s, where we check whether there is a match or not. If there is a match, then we return the shift s; otherwise, we return some invalid value of s, denoting the fact that there is no match. What does this checking take? We need to check the equality of m characters.

So, this particular step takes $O(m)$ time, and the outer loop iterates $O(n-m)$ times in the worst case. So, the time complexity is $O((n-m)m)$, which is $O(nm)$. So, because n is typically much larger than m, okay. So, now we will improve upon this runtime; this is a quadratic running time. So, what will we do?

We will see a brilliant algorithm thanks to Knuth-Morris-Pratt, and this is called the Knuth-Morris-Pratt matcher or Knuth-Morris-Pratt string matcher, often abbreviated as the KMP algorithm. So, let us see that. Knuth-Morris-Pratt. So, we will see. That the worst-case runtime of the KMP algorithm.

is $O(n+m)$ which is optimal up to constant factors because the input length is $n+m$ and any algorithm needs to read the input. So, let us first get the idea and it builds on the nice string matching algorithm, but you see the nice string matching algorithm there are lot of wastage of work. For example, if we find a partial match then we do not use the information that can be conveyed from a partial match in the next iteration and we are forgetting everything about partial match in the naive string matching algorithm and again starting all over the place. So, observe that the naive string matcher does not use partial matches in any way. and the idea of Knuth-Morris-Pratt algorithm is to cleverly use the partial matches. The partial matches using a function called prefix function. So, let us first understand what is prefix function because that is the single most beautiful trick that is used in the KMP algorithm.

So, prefix function is used for the given pattern. So, suppose the given pattern is P 1 to So, for all i or let us use some other notation for all q in between 1 and m we define $\pi(q)$ as follows.

So, π is a function from 1 to q this integers to sorry 1 to m. to integers 0 to $m-1$. So, this is π how is it defined? $\pi(q)$ let me first write in English and then we will formalize it. So, while deciding $\pi(q)$, we look at the first ah q characters of the string P. And we ask that what is the largest strict prefix of $P[1,\ldots,q]$ which matches with the suffix of $P[1,\ldots,q]$ that is $\pi(q)$. The length of the largest strict prefix of the string $P[1,\ldots,q]$ that is also ah suffix of $P[1,\ldots,q]$ formally $\pi(q)$ is the maximum value of the integer k in between 0 to $q-1$ because it needs to be a strict $P[1,\ldots,k]$ the first k characters of the pattern $P[1,\ldots,q]$ matches with the last k characters. So, last character is q let me write here. In particular for k equal to 0 there is always a match and $\pi(q)$ is at least 0 and at most q minus 1. So, let us see an example of pattern and then compute the pi value because this is very important and we need to make sure that we have understood this very clearly.

So, let us take an example. So, suppose the pattern P is A B A B A C A, okay. So, this is how it looks. Now, let us write down the π function. So, pi 1 is always 0 because the empty string is only the strict prefix of a string of length 1.

So, $\pi(1)$ is always 0. To compute $\pi(2)$, I need to look at the string $P[12]$, which is A B, and I ask: what is the length of the longest prefix which is also a suffix? It is 0 here. Now, for $\pi(3)$, I look at the string A B A, and the length of the longest prefix is 1, which is also a suffix. That is, a prefix of length 1 is A, and a suffix of length 1 is A, and they match. So, $\pi(3)$ is 1. Now, let us look at $\pi(4)$.

So, we see that the prefix A B is also a suffix A B, and that is the longest prefix which is also a suffix. So, $\pi(4)$ is 2. For $\pi(5)$, the string is A B A B A. The prefix A matches with the suffix A, but that is not the longest prefix that matches with the suffix of the string. The longest prefix which matches with the suffix is A B A.

The string ABA is a prefix of ABABA and also a suffix of that particular string. So, then $\pi(5)$ is 3. Now $\pi(6)$ because it ends with c and c does not appear anywhere in the first 5 characters $\pi(6)$ is 0 and $\pi(7)$ it ends with a which is also a prefix. So, $\pi(7)$ is 1 ok. So, this is the prefix function.

Now, let us look at the KMP matcher algorithm and assuming that we have already computed the prefix function of the pattern given pattern P and using that we will see how we can find or a matching of P in T the text in $O(n)$ time. And then we will come back and see how we can compute this prefix function π efficiently. So, to begin with.

Let us assume that we have already computed the prefix function pi and then let us look at the Knuth Morris Pratt matcher KMP string matcher which takes and the pattern as input ok. The first step is we need to compute the prefix function ok and then we start our matching q is the currently current shift we are considering.

we are reading the one character at a time in the for loop in the ith iteration I am reading the ith character for i equal to then 1 to n and in q I am storing the partial match that we have got in the $(i-1)$-th iteration. So, initially there was no match. So, q was Otherwise there was a partial match in the last iteration we are we are checking whether can we extend the partial match. So, for that while q greater than 0 and $P[q+1] \neq T[i]$. So, if $P[q+1]$ is equal to $T[i]$ then we are able to extend the partial matching by one more size right. Otherwise if we cannot extend the matching then we use a partial matching of the string $P[1,\ldots,q]$. and that is where the prefix function comes into picture instead of pretending that q size partial match we have got we try to match with a partial match of size $\pi(q)$ and again try to extend ok. So, when we come out of the while loop either q is 0 or $P[q+1]=T[i]$ that means, we can extend.

So, if $P[q+1]=T[i]$, then that means, we are we can extend the current matching by one more length we have a q size partial match. Now, we have a $q+1$ size partial match and after that I check if the partial match length is m or not if we have a partial match of length m that means, it is a match. So, if q equal to m then return $i-m$. So, the matching has started from $i-m+1$ that is the shift ok.

And suppose our job is to get one matching only. So, this is where the loop ends—yeah, this is where the for loop ends. And if you do not find any matching, then we return some invalid shift denoting that fact. That there was no match. So, let us stop here. In the next lecture, we will analyze this algorithm and prove that the worst-case running time of this algorithm is $O(n)$. Observe that if we crudely analyze the runtime, the outer for loop iterates for $O(n)$ iterations, and this inner while loop can iterate for $O(m)$ iterations.

So, a crude bound is $O(mn)$, which is correct, but that is not tight. In the next lecture, we will see that the worst-case running time of this algorithm is actually $O(n)$, okay? So, let us stop here. Thank you.