Lecture Transcript
Rabin-Karp Algorithm

Hi and welcome to this next lecture on Data Structures and Algorithms. We discussed on naive algorithms first string matching which requires one to iterate over all the characters in the input pattern to be matched as well as over all the characters on the text to be matched and in fact we had to redo or repeat a lot of computation. Starting today are approaches that leverage computation that we have already performed to avoid repeated computations and bootstrap around them. The first algorithm we'll discuss as the Rabin-Karp Algorithm. The Rabin-Karp Algorithm basically is hinged upon identifying the vocabulary and the size of the vocabulary around which strings in the input text and the pattern will be constructed. So, suppose the vocabulary was restricted to ten values, so alphabet was 0 to 9. In general, you could have d values.

So, the idea of Rabin-Karp Algorithm is to represent the input sequence or any substring of the input sequence as a digit in the radix-d notation. So basically a string of k consecutive characters represents a length-k decimal number or as a pointed out more generally it represents number in the radix or base d form. Here t was taken to be ten. So to find all occurrences of the patter P in the text T, will compute the hash which is basically number mod q and q is chosen to be a prime of the number p of size m, and compare it with m consecutive digits of T.

Now, why hash? Well, one could do the following one could actually look at the decimal all the radix-d representation for the text say starting at position s and ending at position s+m-1. Compare that with the pattern starting at position one and ending at position m and one could actually check if that the, that is d representation for d for T and P are exactly the same. However this can be expensive. So d the general formula for representing a number the radix-d notation is as follows. So let's say ts is computed or defined to be T, starting at position [s+1, and ending at position s+m]. So, ts will correspond to T[s + m] + 10 times (T(s + m - 1) + 10 times T[s + m - 2] and so on until T(s+1). One can do likewise for P. So p is basically P[m] + 10 times (P(m-1) + 10 times (P(m - 2)) until p(1). So computing the t and p, ts and p can be expensive. In fact, it might it not be even possible to fit T, S or P into one computer word. So hash comes to our rescue here. The hash which is of the number that is ts modulo sum prime q would be such that it fits as one word in the computer and likewise for P. So P is chosen in order to make this number mod q small enough. However, this comes at a price. Even if two hash numbers match, which means even if p mod q = t(sub)s mode q. It is not necessary that p and t(sub)s are same . So even if the two hash numbers match, you will still need to go back and check for the digits of T and P being the same. So there is an additional overhead that one incurs for having used this hash function.

Now, one can provide a counter example, such as a raised to n and supposed your pattern was just a raised to m. One can easily see that they will be a match, potential match of the hash functions at every position of T and this will lead to the checking for the equivalence of digits of T and P at every position. So one could compute the worst case running time to be theta of (n-m+1), this is the time required to scan T, times m. So the worst case, it could be order of n, m and therefore expensive. However, in most practical cases when the number of hash matches are constant. One will find that the average case complexity is very reasonable. So we'll look at the average case complexity once we discuss the algorithm itself. So once you have doubly check for the match of T, a segment of T with P. You report the occurrence of this pattern P in the text T. So here is an illustration. Here we looking at an alphabet set sigma which is bringing from 0 to 9. So we are basically looking at decimal representations.

So, the choice of q in this case is a prime number 17. So we look at the pattern P, the decimal representation mod 17 is 15. Now, suppose we had to match this against this long text. You would start from the leftmost end, look at the first 5 digits, compute mod 17 will it turns out to be 1. So, if P mod q was not equal to ts mod q, one can be very confident that P is certainly not equal to ts. So we can proceed, we look at the next five digits, mod 17 is 15. Now, this is a valid match. So what we find here is the P mod q equals the ts+1 mod q. Now, does it mean P = ts+1? No. So now you need to verify that P indeed equals ts+1 and was this correct? 8 4 7 2 6 so that was indeed a valid match, so we flag a valid match. Next 7 2 6 3 9, well, you get the same mod 17 value here. So we find that P mod q indeed = ts+2, ts+3 mod q, but P is not equal to ts+3. So there was a false alarm that the hash value collision, I later ask too. So this is the price we need to pay for computing hash function.

Now, how many mistakes do we expect to see? So we can expect the number of such spurious hits to be, basically the order of the number of digits the length of the input text divided by the prime q. So this number of false alarms or spurious hits, they can be expected the order of n/q. Continuing we find invalid match. So is there a way that we could compute the hash value leveraging computations from the past. So 3 8 4 7 2 mod 17 is 1. Can that we used to discover that 8 4 7 2 6 with the additional digit 6 mod 17 is actually 15. So we have an old high order digit 3 that disappears and the new low-order digit 6 that appears. So one can update the mod value using the Horner's rule as follows. So the decimal value ts+1 can be obtained by first of all subtracting from ts 3 this the largest digit multiplied by the number of the base exponentiated by the number of remaining digits. So what does that mean? ts-10 raised to m-1 times the value at position s+1. However, after the subtraction I need to do a left shift in order to insert 6. So this left shift can be achieved by multiplying by the same base 10, and to this we can add the new digit T[s+m+1]. So this operation corresponds to removal of 3 digit corresponding to the largest value and pre multiplication with 10 corresponds to left shifting what remains. Finally, addition of T at s+m+1 corresponds to adding a digit at the least significant value.

Now all these does it carry forwarded to mod operations? So, recall from modular arithmetic, we have some properties from modular arithmetic for addition and multiplication. So in multiplication (a mod q) times (b mod q) is equivalent to ab mod q one can write this in terms of the normal equality as follows. So this just means that this multiplication mod q itself is a same as (ab) mod q. So making use of modular arithmetic properties one could update the mod mod for the new number 84726 as follows. One can actually take mod of the terms on the right-hand side. and

what we note here is we have 10 raised to m-1 or in general this could be d raised to m -1. This might be expensive to compute at every step. However, by virtue of modular arithmetic one could actually compute h as 10 raised to m-1 mod q.

This can be a substitute the h can be precomputed and the smaller value can be substituted to be able to do this computation in more reasonable time. So this precomputation involves computing p mod q and t0 mod q. t0 you might recall corresponds to the string starting at position 0 and ending at position m-1 in t. So begin with we compute t0 and p and as already motivated we compute rather precompute d raised to m-1 mod q. We have made an assumption here that the text T consists of a vocabulary sigma which is of size d. So we're able to create a one to one mapping from every character in sigma to one of the digits 0 to d or rather 0 to d-1. So, once we have done this precomputation we only need to make use of with the modified Horner's rule with the modulo and compute the mod for subsequent sequences. So this is what we do, we look for a match of p to ts and we do this for every position s if indeed there is a match let me basically ensure that the two strings are the same. We ensure that P position 0 to m-1 indeed is equal to T starting at position s and going until position s+m-1.

So, if j equals m if index j finds a match for all the m-1 positions in both T and P then we know that the shift at s is valid rather there was a match at a position starting at s in t. You keep doing this and for s less than n-m i.e. for s that ends well before m characters of t we're going to dynamically update ts +1 and we do that using the properties and modular arithmetic that we already discussed. So, recall that this h was the general d raised to m-1 mod q. ts itself was obtained after mod q operation. So T was T going from s to s+m-1 mod q. You, of course, perform this multiplication and addition and again check for mod q match. If there was a match here you would go back and check for characterwise equality. So analysis, the precomputation which involves scan of the m positions of P and T is basically an order m operation. The matching of every position and T the sub strings originating at every position and T with p is going to involve a scan over the entire array T. That will be n - m+1 positions in T. Now, if there were C of such hits which we've spurious or what we refer to earlier as false alarms the expected matching time going to this false alarms will basically of the order of n-m+1 which we already accounted for plus c times m because with every spurious hit you need to do a scan over both P and T.

We already seen the worst case the worst case was when T was a raised to n and P was a occurring m times and this led to an order n-m+1 times m. However, the average case is actually better so on an average one can expect order of n/q spurious hits. Basically that an arbitrary T will be equivalent to P modulo q can be estimated to be 1/q. And there are only order n positions so this leads to order of n/q spurious hits and one can show that the running time in the average case is basically order of n. What we see here below is the worst case running time and this is illustrated through this example of T being sequence of n's and P being the sequence of m's. That is when we incur n-m+1 spurious hits.

Thank you.