



Algorithms:

COMP3121/3821/9101/9801

Aleks Ignjatović

School of Computer Science and Engineering
University of New South Wales

LECTURE 8: STRING MATCHING ALGORITHMS

String Matching algorithms

- Assume that you want to find out if a string $B = b_1b_2 \dots b_m$ appears as a (contiguous) substring of a much longer string $A = a_1a_2 \dots a_n$.
- The “naive” string matching algorithm does not work well if B is much longer than what can fit in a single register; we need something cleverer.
- Assume that strings A and B are in an alphabet \mathcal{A} with d many symbols in total.

Rabin - Karp Algorithm

- We now show how hashing can be combined with recursion to produce an efficient string matching algorithm.
- We compute the hash value for the string $B = b_1b_2 \dots b_m$ in the following way.
- We have assumed that the underlying alphabet has d many symbols in total. Thus, we can identify each string with a sequence of integers by mapping each symbol s_i into a corresponding integer i :

$$\mathcal{A} = \{s_0, s_1, s_2, \dots, s_{d-1}\} \longrightarrow \{0, 1, 2, \dots, d-1\}$$

- To any string $B = b_1b_2 \dots b_m$ we can now associate an integer whose digits in base d are integers corresponding to each symbol in B :

$$h(B) = h(b_1b_2 \dots b_m) = d^{m-1}b_1 + d^{m-2}b_2 + \dots + d \cdot b_{m-1} + b_m$$

- This can be done efficiently using the Horner's rule:

$$h(B) = b_m + d(b_{m-1} + d(b_{m-2} + d(b_{m-3} + \dots + d(b_2 + d \cdot b_1)))) \dots$$

- Next choose a large prime number p such that $(d+1)p$ fits in a single register and define the hash value of B as $H(B) = h(B) \bmod p$.

Rabin - Karp Algorithm

- Recall that $A = a_1 a_2 a_3 \dots a_s a_{s+1} \dots a_{s+m-1} \dots a_N$ where $N \gg m$.
- We want to find efficiently all s such that strings $a_s a_{s+1} \dots a_{s+m-1}$ and $b_1 b_2 \dots b_m$ are equal.
- For each contiguous substring $A_s = a_s a_{s+1} \dots a_{s+m-1}$ of string A we also compute its hash value as

$$H(A_s) = (d^{m-1}a_s + d^{m-2}a_{s+1} + \dots + d^1a_{s+m-2} + a_{s+m-1}) \bmod p$$

- We can now compare the hash values $H(B)$ and $H(A_s)$ and do a symbol-by-symbol matching only if $H(B) = H(A_s)$.
- Clearly, such an algorithm would be faster than the naive symbol-by-symbol comparison only if we can compute the hash values of substrings A_s faster than what it takes to compare the strings B and A_s character by character.
- This is where recursion comes into play: we do not have compute the hash value $H(A_{s+1})$ of $A_{s+1} = a_{s+1} a_{s+2} \dots a_{s+m}$ “from scratch”, but we can compute it efficiently from the hash value $H(A_s)$ of $A_s = a_s a_{s+1} \dots a_{s+m-1}$ as follows.

Rabin - Karp Algorithm

Since

$$H(A_s) = (d^{m-1}a_s + d^{m-2}a_{s+1} + \dots d^1a_{s+m-2} + a_{s+m-1}) \bmod p$$

we obtain

$$\begin{aligned}(d \cdot H(A_s)) \bmod p &= \\&= (d^m a_s + d^{m-1} a_{s+1} + \dots d \cdot a_{s+m-1}) \bmod p \\&= ((d^m a_s) \bmod p + (d^{m-1} a_{s+1} + \dots d^2 a_{s+m-2} + d a_{s+m-1} + a_{s+m}) \bmod p - a_{s+m}) \bmod p \\&= ((d^m a_s) \bmod p + H(A_{s+1}) - a_{s+m}) \bmod p\end{aligned}$$

Consequently, $H(A_{s+1}) = (d \cdot H(A_s) - (d^m \bmod p)a_s + a_{s+m}) \bmod p$.

- Note that in the expression

$$H(A_{s+1}) = (d \cdot H(A_s) - (d^m \bmod p)a_s + a_{s+m}) \bmod p$$

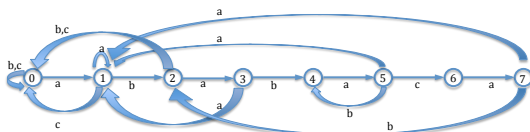
the value $d^m \bmod p$ can be precomputed and is smaller than p .

- Thus, since we chose p such that $(d+1)p$ fits in a register, all the values and the intermediate results for the above expression also fit in a single register.
- The value of $H(A_s)$ can be computed in constant time independent of the length of the strings A and B .

String matching finite automata

- A string matching finite automaton for a string S with k symbols has $k + 1$ many states $0, 1, \dots, k$ which correspond to the number of characters matched thus far and a transition function $\delta(s, c)$ where s is a state and c is a character, given by a table.
- To make things easier to describe, we consider the string $S = ababaca$. The table defining $\delta(s, c)$ would then be

state	input			
	a	b	c	
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	



state transition diagram for string *ababaca*

String matching with finite automata

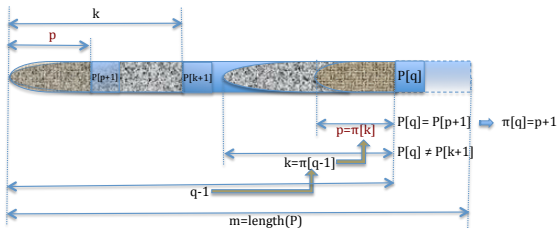
- How do we compute the transition function δ , i.e., how do we fill the table?
- Let P_k denote the prefix of the string P consisting of the first k characters of string P .
- If we are at a state k this means that so far we have matched the prefix P_k ; if we now see an input character a , then $\delta(k, a)$ is the largest m such that the prefix P_m of string P is the suffix of the string $P_k a$.
- Thus, if a happens to be $P[k + 1]$, then $m = k + 1$ and so $\delta(k, a) = k + 1$ and $P_k a = P_{k+1}$.

String matching with finite automata

- We can get by without precomputing $\delta(k, a)$ but instead compute it “on the fly”.
- We do that by matching the string against itself: we can recursively compute a function $\pi(k)$ which for each k returns the largest integer m such that the prefix P_m of P is a proper suffix of P_k .

The Knuth-Morris-Pratt algorithm

```
1: function  
   Compute - Prefix - Function( $P$ )  
2:    $m \leftarrow \text{length}[P]$   
3:   let  $\pi[1..m]$  be a new  
   array  
4:    $\pi[1] = 0$   
5:    $k = 0$   
6:   for  $q = 2$  to  $m$  do  
7:     while  $k > 0$  and  
        $P[k+1] \neq P[q]$   
8:        $k = \pi[k]$   
9:     if  $P[k+1] == P[q]$   
10:       $k = k + 1$   
11:     $\pi[q] = k$   
12:  end for  
13:  return  $\pi$   
14: end function
```



Assume that length of P is m and that we have already found that $\pi[q-1] = k$; to compute $\pi[q]$ we check if $P[q] = P[k+1]$; if it is not; then $\pi[q] \neq k+1$ and we find $\pi[k] = p$; if now $P[q] = P[k+1]$ then $\pi[q] = p+1$.

Example:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$B =$	1	2	3	1	2	4	1	2	3	1	2	3	1	2	4	1	2	3	1	2	4

- $\pi[1] = 0, \pi[2] = 0, \pi[3] = 0, \pi[4] = 1, \pi[5] = 2, \pi[6] = 0,$
 $\pi[7] = 1, \pi[8] = 2, \pi[9] = 3, \pi[10] = 4, \pi[11] = 5, \pi[12] = 3 \dots$
- Complete the above calculation of $\pi[k]$ for $k = 13$ to $k = 21$.

The Knuth-Morris-Pratt algorithm

- We can now do our search for string P in a longer string T :

```
1: function KMP – Matcher( $T, P$ )
2:    $n \leftarrow \text{length}[T]$ 
3:    $m \leftarrow \text{length}[P]$ 
4:    $\pi = \text{Compute – Prefix – Function}(P)$ 
5:    $q = 0$ 
6:   for  $i = 2$  to  $n$  do
7:     while  $q > 0$  and  $P[q + 1] \neq T[i]$ 
8:        $q = \pi[q]$ 
9:     if  $P[q + 1] == T[i]$ 
10:       $q = q + 1$ 
11:     if  $q == m$ 
12:       print pattern occurs with shift  $i - m$ 
13:      $q = \pi[q]$ 
14:   end for
15: end function
```