School of Computer Science and En
University of New South Wales

9. STRING MATCHING ALGORITHMS

- Assume that you want to find out if a string $B = b_0 b_1 \ldots b_{m-1}$ appears as a (contiguous) substring of a much longer string $A = a\ \ a_1 \ldots a_{n-1}$.

- Assume that you want to find out if a string $B = b_0 b_1 \ldots b_{m-1}$ appears as a (contiguous) substring of a much longer string $A = a\; a_1 \ldots a_{n-1}$.

- The B s much long

- Assume that you want to find out if a string $B = b_0 b_1 \ldots b_{m-1}$ appears as a (contiguous) substring of a much longer string $A = a_0 a_1 \ldots a_{n-1}$.

- The [illegible] B is much long[illegible]

- We now show how hashing can be combined with re[illegible] an efficient string matching algorithm.

Rabin - Karp Algorithm

- We compute a hash value for the string $B = b_0 b_1 b_2 \ldots b_m$ in the following way.

Rabin - Karp Algorithm

- We compute a hash value for the string $B = b_0 b_1 b_2 \dots b_m$ in the following way.
- We will assume that strings A and B are in an alphabet $\mathcal{A}$ with d many symbols in total.

- We compute a hash value for the string $B = b_0 b_1 b_2 \ldots b_m$ in the following way.
- We will assume that strings A and B are in an alphabet $\mathcal{A}$ with d many symbols in total.
- Thus, we can identify each string with a sequence of integers by mapping each sym

Rabin - Karp Algorithm

- We compute a hash value for the string $B = b_0 b_1 b_2 \ldots b_m$ in the following way.
- We will assume that strings A and B are in an alphabet $\mathcal{A}$ with d many symbols in total.

- Thus, we can identify each string with a sequence of integers by mapping each sym

- To any string $B = b_0 b_1 \ldots b_{m-1}$ we can now associate an integer whose digits in base d are integers corresponding to each symb

$$h(B) = h(b_0 b_1 b_2 \ldots b_m) = d^{m-1} b_0 + \quad {}_{-1}$$

Rabin - Karp Algorithm

- We compute a hash value for the string $B = b_0 b_1 b_2 \ldots b_m$ in the following way.
- We will assume that strings A and B are in an alphabet $\mathcal{A}$ with d many symbols in total.

- Thus, we can identify each string with a sequence of integers by mapping each sym

- To any string $B = b_0 b_1 \ldots b_{m-1}$ we can now associate an integer whose digits in base d are integers corresponding to each symb

$$h(B) = h(b_0 b_1 b_2 \ldots b_m) = d^{m-1} b_0 + \ldots {}_{-1}$$

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

$$h(B) = b_{m-1} + d(b_{m-2} + d(b_{m-3} + d(b_{m-4} + \ldots + d(b_1 + d \cdot b_0))) \ldots)$$

- We compute a hash value for the string $B = b_0 b_1 b_2 \ldots b_m$ in the following way.
- We will assume that strings A and B are in an alphabet $\mathcal{A}$ with d many symbols in total.

- Thus, we can identify each string with a sequence of integers by mapping each sym

- To any string $B = b_0 b_1 \ldots b_{m-1}$ we can now associate an integer whose digits in base d are integers corresponding to each symb

$$h(B) = h(b_0 b_1 b_2 \ldots b_m) = d^{m-1} b_0 + \ldots -1$$

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

$$h(B) = b_{m-1} + d(b_{m-2} + d(b_{m-3} + d(b_{m-4} + \ldots + d(b_1 + d \cdot b_0))) \ldots)$$

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

Rabin - Karp Algorithm

- Recall that $A = a_0a_1a_2a_3\ldots\ldots a_sa_{s+1}\ldots a_{s+m-1}\ldots\ldots a_{N-1}$ where $N >> m$.

Rabin - Karp Algorithm

- Recall that $A = a_0 a_1 a_2 a_3 \ldots \ldots a_s a_{s+1} \ldots a_{s+m-1} \ldots \ldots a_{N-1}$ where $N >> m$.
- We want to find efficiently all s such that the string of length m of the form $a_s a_{s+1} \ldots a_{s+m-1}$ and string $b_0 b_1 \ldots b_{m-1}$ are equal.

Rabin - Karp Algorithm

- Recall that $A = a_0 a_1 a_2 a_3 \ldots\ldots a_s a_{s+1} \ldots a_{s+m-1} \ldots\ldots a_{N-1}$ where $N >> m$.
- We want to find efficiently all s such that the string of length m of the form $a_s a_{s+1} \ldots a_{s+m-1}$ and string $b_0 b_1 \ldots b_{m-1}$ are equal.
- For each contiguous substring $A_s = a_s a_{s+1} \ldots a_{s+m-1}$ of string A we also compute its hash value as

$$) \bmod p$$

- Recall that $A = a_0 a_1 a_2 a_3 \ldots\ldots a_s a_{s+1} \ldots a_{s+m-1} \ldots\ldots a_{N-1}$ where $N >> m$.
- We want to find efficiently all s such that the string of length m of the form $a_s a_{s+1} \ldots a_{s+m-1}$ and string $b_0 b_1 \ldots b_{m-1}$ are equal.
- For each contiguous substring $A_s = a_s a_{s+1} \ldots a_{s+m-1}$ of string A we also compute its hash value as

$$) \bmod p$$

- We c $_s$ symbol-by-symbol matching only if $H(B) =$

- Recall that $A = a_0a_1a_2a_3\ldots\ldots a_sa_{s+1}\ldots a_{s+m-1}\ldots\ldots a_{N-1}$ where $N >> m$.
- We want to find efficiently all s such that the string of length m of the form $a_sa_{s+1}\ldots a_{s+m-1}$ and string $b_0b_1\ldots b_{m-1}$ are equal.
- For each contiguous substring $A_s = a_sa_{s+1}\ldots a_{s+m-1}$ of string A we also compute its hash value as

$$) \bmod p$$

- We c s symbol-by-symbol matching only if $H(B) =$
- Clearly, such an algorithm would be faster than the nai comparison only if we can compute the hash values of su ster than what it takes to compare strings B and A_s character by character.

Rabin - Karp Algorithm

- Recall that $A = a_0 a_1 a_2 a_3 \ldots \ldots a_s a_{s+1} \ldots a_{s+m-1} \ldots \ldots a_{N-1}$ where $N >> m$.
- We want to find efficiently all s such that the string of length m of the form $a_s a_{s+1} \ldots a_{s+m-1}$ and string $b_0 b_1 \ldots b_{m-1}$ are equal.

- For each contiguous substring $A_s = a_s a_{s+1} \ldots a_{s+m-1}$ of string A we also compute its hash value as

$$) \bmod p$$

- We c ... s symbol-by-symbol matching only if $H(B) =$

- Clearly, such an algorithm would be faster than the nai ... comparison only if we can compute the hash values of su ... ster than what it takes to compare 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 $A_{s+1} = a_{s+1} a_{s+2} \ldots a_{s+m}$ "from scratch", but we can compute it efficiently from the hash value $H(A_s)$ of $A_s = a_s a_{s+1} \ldots a_{s+m-1}$ as follows.

Since

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

Since

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

by multipl

$d \cdot H(A_s))$ m

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

$= (d^m a_s + (d^{m-1} a_{s+1} + \ldots d^2 a_{s+m-2} + d\, a_{s+m}$ $) \bmod p$

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

- Consequently,

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

Rabin - Karp Algorithm

- Consequently,

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

- Note that

$$(d^m a_s) \mod p = ((d^m \mod p) a_s) \mod p$$

and t

Rabin - Karp Algorithm

- Consequently,

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

- Note that

$$(d^m a_s) \bmod p = ((d^m \bmod p) a_s) \bmod p$$

and t

- Als

- Consequently,

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

- Note that

$$(d^m a_s) \bmod p = ((d^m \bmod p) a_s) \bmod p$$

and t

- Als

- Thus, since $H(A_s) < p$ we obtain

$$d \cdot H(A_s) + (-d^m a_s + a_{s+}$$

Rabin - Karp Algorithm

- Consequently,

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

- Note that

$$(d^m a_s) \mod p = ((d^m \mod p) a_s) \mod p$$

and t

- Als

- Thus, since $H(A_s) < p$ we obtain

$$d \cdot H(A_s) + (-d^m a_s + a_{s+}$$

- 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.

- Consequently,

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

- Note that

$$(d^m a_s) \mod p = ((d^m \mod p) a_s) \mod p$$

and t

- Als

- Thus, since $H(A_s) < p$ we obtain

$$d \cdot H(A_s) + (-d^m a_s + a_{s+}$$

- 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.

- Thus, for every s except $s = 0$ the value of $H(A_s)$ can be computed in constant time independent of the length of the strings A and B.

- Thus, we first compute $H(B)$ and $H(A_0)$ using the Horner's rule.

- Thus, we first compute $H(B)$ and $H(A_0)$ using the Horner's rule.
- Subsequent values of $H(A_s)$ for $s > 0$ are computed in constant time using the above recursion.

Rabin - Karp Algorithm

- Thus, we first compute $H(B)$ and $H(A_0)$ using the Horner's rule.
- Subsequent values of $H(A_s)$ for $s > 0$ are computed in constant time using the above recursion.
- $H($ A_s and B a equ

- Thus, we first compute $H(B)$ and $H(A_0)$ using the Horner's rule.

- Subsequent values of $H(A_s)$ for $s > 0$ are computed in constant time using the above recursion.

- $H($ … A_s and B a … equ …

- Since p was chosen large, the false positives whe … but $A_s \neq B$ are very unlikely, which makes the algo …

- Thus, we first compute $H(B)$ and $H(A_0)$ using the Horner's rule.

- Subsequent values of $H(A_s)$ for $s > 0$ are computed in constant time using the above recursion.

- $H($ … A_s and B a … equ

- Since p was chosen large, the false positives whe … but $A_s \neq B$ are very unlikely, which makes the algo

- However, as always when we use hashing, we cann … case performance.

- Thus, we first compute $H(B)$ and $H(A_0)$ using the Horner's rule.

- Subsequent values of $H(A_s)$ for $s > 0$ are computed in constant time using the above recursion.
- $H($ … A_s and B a … equ

- Since p was chosen large, the false positives whe … but $A_s \neq B$ are very unlikely, which makes the algo

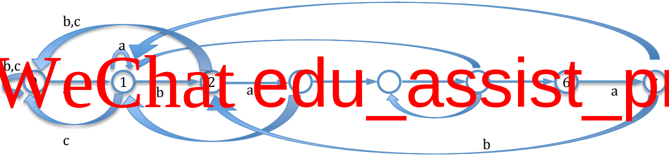
- However, as always when we use hashing, we cann … case performance.
- So we now look for algorithms whose worst case performance can be guaranteed.

- A string matching finite automaton for a string S with k symbols has $k+1$ many states $0, 1, \ldots k$ 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 red at the moment.

- A string matching finite automaton for a string S with k symbols has $k+1$ many states $0, 1, \ldots k$ 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 read at the moment.
- We first look at the case when such $\delta(s, c)$ is given by a pre-constructed table.

- A string matching finite automaton for a string S with k symbols has $k+1$ many states $0, 1, \ldots k$ 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 read at the moment.
- We first look at the case when such $\delta(s, c)$ is given by a pre-constructed table.
- To make things easier to describe, we consider the string $S = ababaca$. The table defin

state	a			
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*

- How do we compute the transition function δ, i.e., how do we fill the table?

- How do we compute the transition function δ, i.e., how do we fill the table?

- Let B_k denote the prefix of the string B consisting of the first k cha

- How do we compute the transition function δ, i.e., how do we fill the table?

- Let B_k denote the prefix of the string B consisting of the first k cha

- If we B_k; if m such that m k .

- How do we compute the transition function δ, i.e., how do we fill the table?

- Let B_k denote the prefix of the string B consisting of the first k cha

- If we B_k; if m such that m k.

- Thus, if a happens to be $B[k+1]$, then $k+1$ and $B_k a = B_{k+1}$.

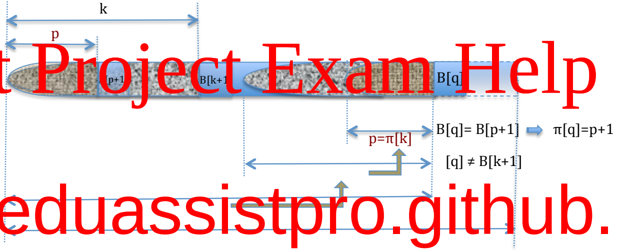
- How do we compute the transition function δ, i.e., how do we fill the table?

- Let B_k denote the prefix of the string B consisting of the first k cha

- If we B_k; if m such that m k .

- Thus, if a happens to be $B[k+1]$, then $k+1$ and $B_k a = B_{k+1}$.

- 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 B_m of B is a proper suffix of B_k.



function Compute − Prefix − Function(B)
  $m \leftarrow \text{length}[B]$
  let $\pi[1..m]$ be a new array
  $\pi[1] = 0$
  $k = 0$
  for q
    w
      $B[k+1] \neq B[q]$
    $k = \pi[k]$
    if $B[k+1] == B[q]$
      $k = k + 1$
    $\pi[q] = k$
  end for
  return π
end function

Assume that len… we have already f… k; to compute $\pi[q]$ we check if $B[q] = B[k+1]$; if true then $\pi[q] = k+1$; if not true then we find $\pi[k] = p$; if now $B[q] = B[p+1]$ then $\pi[q] = p+1$.

- We can now do our search for string B in a longer string A:


```
function KMP − Matcher(A, B)
    n    length[A]
    m
    π
    q
    fo
        while q > 0 and B[q + 1] ≠ A[i]
        q = π[q]
        if  B[q + 1] == A[i]
            q = q + 1
        if  q == m
            print pattern occurs with shift i − m
            q = π[q]
    end for
end function
```

Looking for imperfect matches

- Sometimes we are not interested in finding just the prefect matches, but also in matches that might have a few errors, such as a few insertions, deletions and replacements.

- So ass
  $A =$
  $B =$ nterested
  in fin

- Idea: split B into $k + 1$ consecutive subsequen
  length. Then any match in A with at most
  subsequence which is a perfect match for a subsequenc k
  for all perfect matches for all of $k + 1$ subseq e
  test by brute force if the remaining parts of B have sufficient number of matches in the appropriate parts of A.

PUZZLE!!

On a rectangular table there are 25 non-overlapping round coins of equal size

coin with

falling off t

within the table). Show that it is possible to complete

with 100 coins (of course with overlapping of coins).