CSE408
String Matching Algorithm

Lecture # 5&6

String Matching Problem

Motivations: text-editing, pattern matching in DNA sequences

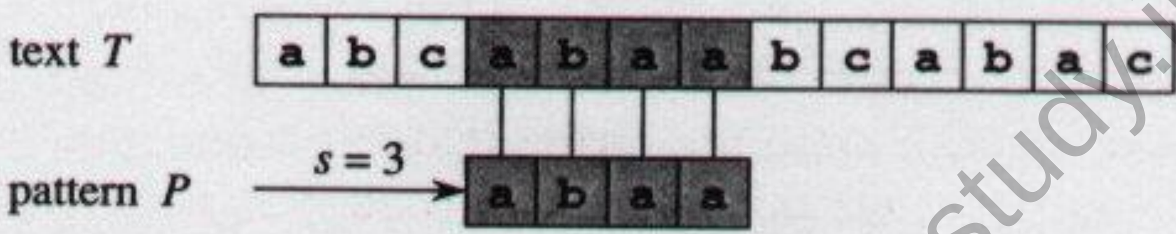


Figure 32.1 The string-matching problem. The goal is to find all occurrences of the pattern P = abaa in the text T = abcabaabcabac. The pattern occurs only once in the text, at shift $s = 3$. The shift $s = 3$ is said to be a valid shift. Each character of the pattern is connected by a vertical line to the matching character in the text, and all matched characters are shown shaded.

Text: array T[1...n] $n > m$ **Pattern**: array P[1...m]

Array Element: Character from finite alphabet Σ

Pattern P occurs with shift s in T if P[1...m] = T[s+1...s+m]

$0 \le s \le n - m$

String Matching Algorithms

- Naive Algorithm
  - Worst-case running time in $O((n-m+1)\ m)$
- Rabin-Karp
  - Worst-case running time in $O((n-m+1)\ m)$
  - Better than this on average and in practice
- Knuth-Morris-Pratt
  - Worst-case running time in $O(n + m)$

Notation & Terminology

- Σ* = set of all finite-length strings formed using characters from alphabet Σ
- Empty string: ε
- |x| = length of string x
- w is a prefix of x: w x
- w is a suffix of x: w x
- prefix, suffix are *transitive*

ab abcca

cca abcca

Naive String Matching

NAIVE-STRING-MATCHER(T, P)

```
n ← length[T]
m ← length[P]
for s ← 0 to n − m
      do if P[1..m] = T[s + 1..s + m]
            then print "Pattern occurs with shift" s
```

worst-case running time is in $\Theta((n-m+1)m)$

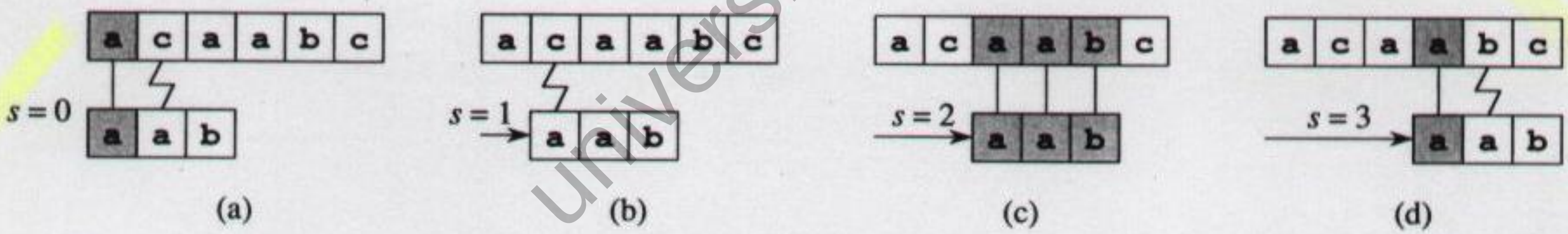


Figure 32.4 The operation of the naive string matcher for the pattern P = aab and the text T = acaabc. We can imagine the pattern P as a "template" that we slide next to the text. Parts (a)–(d) show the four successive alignments tried by the naive string matcher. In each part, vertical lines connect corresponding regions found to match (shown shaded), and a jagged line connects the first mismatched character found, if any. One occurrence of the pattern is found, at shift $s = 2$, shown in part (c).

Rabin-Karp Algorithm

- Assume each character is digit in radix-d notation (e.g. d=10)
- p = decimal value of pattern
- t_s = decimal value of substring T[s+1..s+m] for s = 0,1...,n-m
- Strategy:
  - compute p in O(m) time (which is in O(n))
  - compute all t_i values in total of O(n) time
  - find all valid shifts s in O(n) time by comparing p with each t_s
- Compute p in O(m) time using Horner's rule:
  - p = P[m] + d(P[m-1] + d(P[m-2] + ... + d(P[2] + dP[1])))
- Compute t_0 similarly from T[1..m] in O(m) time
- Compute remaining t_i's in O(n-m) time
  - $t_{s+1} = d(t_s - d^{m-1}T[s+1]) + T[s+m+1]$

Rabin-Karp Algorithm

But... p, t_s may be large, so use mod

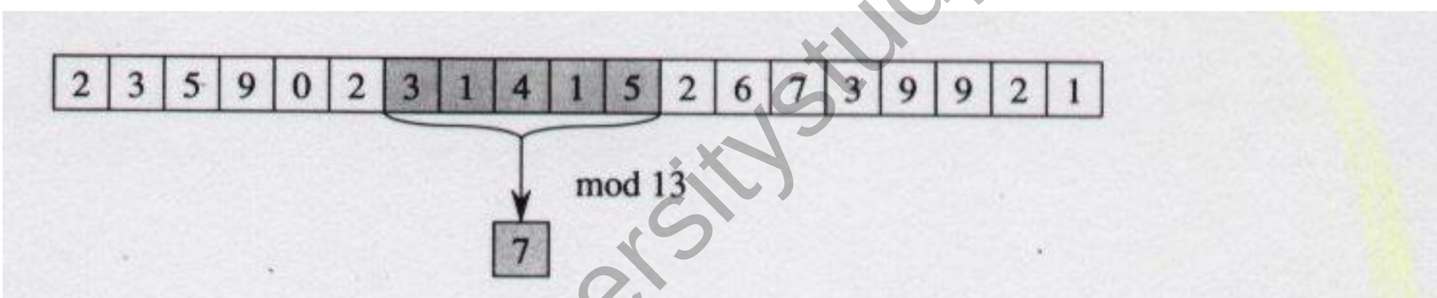


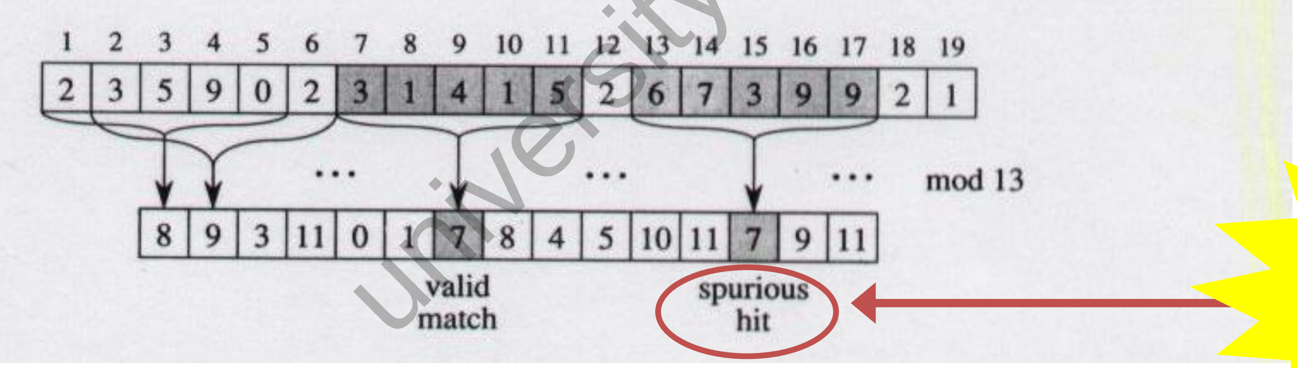
Figure 32.5 The Rabin-Karp algorithm. Each character is a decimal digit, and we compute values modulo 13. **(a)** A text string. A window of length 5 is shown shaded. The numerical value of the shaded number is computed modulo 13, yielding the value 7.

Rabin-Karp Algorithm (continued)

But...

$t_{s+1} = d(t_s - d^{m-1}T[s+1]) + T[s+m+1]$

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q \, , \qquad (34.2)$$

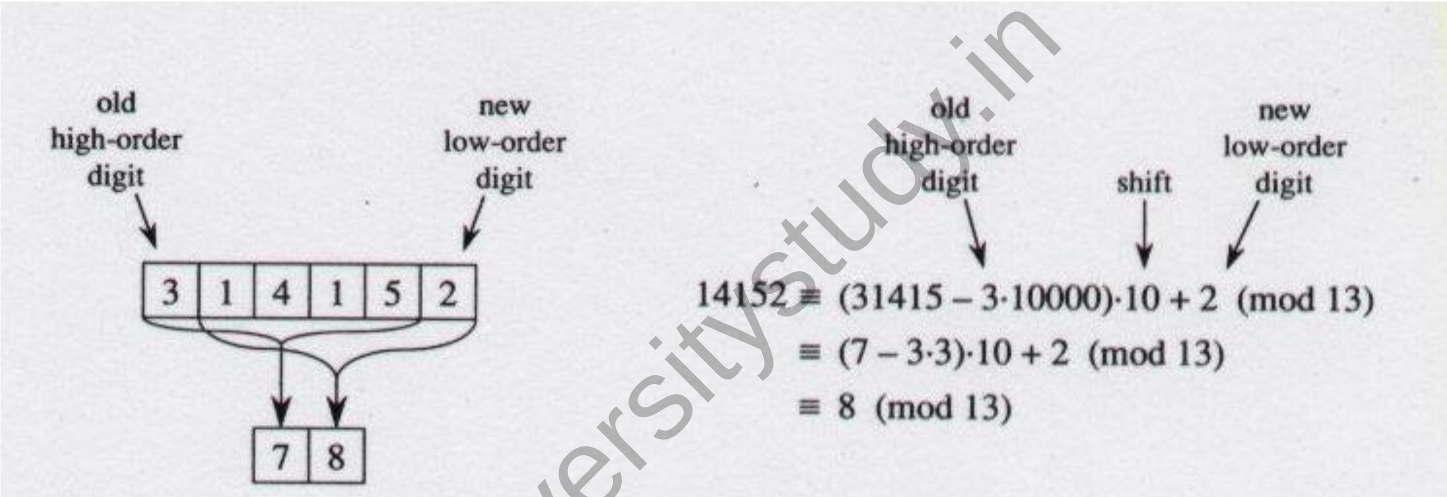


p = 31415

spurious hit

(b) The same text string with values computed modulo 13 for each possible position of a length-5 window. Assuming the pattern $P = 31415$, we look for windows whose value modulo 13 is 7, since $31415 \equiv 7 \pmod{13}$. Two such windows are found, shown shaded in the figure. The first, beginning at text position 7, is indeed an occurrence of the pattern, while the second, beginning at text position 13, is a spurious hit.

Rabin-Karp Algorithm (continued)



[redacted] **(c)** Computing the value for a window in constant time, given the value for the previous window. The first window has value 31415. Dropping the high-order digit 3, shifting left (multiplying by 10), and then adding in the low-order digit 2 gives us the new value 14152. All computations are performed modulo 13, however, so the value for the first window is 7, and the value computed for the new window is 8.

Rabin-Karp Algorithm (continued)

RABIN-KARP-MATCHER(T, P, d, q) — d is radix q is modulus

1 $n \leftarrow length[T]$
2 $m \leftarrow length[P]$
3 $h \leftarrow d^{m-1} \bmod q$ — high-order digit position for m-digit window
4 $p \leftarrow 0$
5 $t_0 \leftarrow 0$
6 **for** $i \leftarrow 1$ **to** m — Preprocessing
7   **do** $p \leftarrow (dp + P[i]) \bmod q$
8         $t_0 \leftarrow (dt_0 + T[i]) \bmod q$
9 **for** $s \leftarrow 0$ **to** $n - m$
10   **do if** $p = t_s$
11   **then if** $P[1..m] = T[s+1..s+m]$ — rule out spurious hit
12   **then** "Pattern occurs with shift" s
13   **if** $s < n - m$
14   **then** $t_{s+1} \leftarrow (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$

$\Theta(m)$ in $\Theta(n)$ (line 3)

$\Theta(m)$ (lines 6–8)

Matching loop invariant: when line 10 executed $t_s = T[s+1..s+m] \bmod q$

$\Theta((n-m+1)m)$ (lines 9–14) — Try all possible shifts

$\Theta(m)$ (lines 10–14)

What input generates worst case?

worst-case running time is in $\Theta((n-m+1)m)$

Rabin-Karp Algorithm (continued)

d is radix   q is modulus

```
RABIN-KARP-MATCHER(T, P, d, q)
1  n ← length[T]
2  m ← length[P]
3  h ← d^(m-1) mod q
4  p ← 0
5  t_0 ← 0
6  for i ← 1 to m
7       do p ← (dp + P[i]) mod q
8          t_0 ← (dt_0 + T[i]) mod q
9  for s ← 0 to n − m
10      do if p = t_s
11            then if P[1..m] = T[s+1..s+m]
12                   then "Pattern occurs with shift" s
13         if s < n − m
14            then t_{s+1} ← (d(t_s − T[s+1]h) + T[s+m+1]) mod q
```

$\Theta(m)$ in $\Theta(n)$ → line 3

high-order digit position for m-digit window

Preprocessing

$\Theta(m)$ (lines 6–8)

Matching loop invariant: when line 10 executed t_s=T[s+1..s+m] mod q

rule out spurious hit

Worst Case

$\Theta((n-m+1)m)$

Try all possible shifts

$\Theta(m)$

Average Case

Assume reducing mod q is like random mapping from Σ^* to Z_q

Estimate (chance that t_s= p mod q) = 1/q → # spurious hits is in O(n/q)

Expected matching time = O(n) + O(m(v + n/q))   (v = # valid shifts)

If v is in O(1) and q >= m → average-case running time is in O(n+m)

The Knuth-Morris-Pratt Algorithm

Knuth, Morris and Pratt proposed a linear time algorithm for the string matching problem.

A matching time of O(n) is achieved by avoiding comparisons with elements of ‘S’ that have previously been involved in comparison with some element of the pattern ‘p’ to be matched. i.e., backtracking on the string ‘S’ never occurs

Components of KMP algorithm

- <u>The prefix function, Π</u>

The prefix function,Π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern ‘p’. In other words, this enables avoiding backtracking on the string ‘S’.

- <u>The KMP Matcher</u>

With string ‘S’, pattern ‘p’ and prefix function ‘Π’ as inputs, finds the occurrence of ‘p’ in ‘S’ and returns the number of shifts of ‘p’ after which occurrence is found.

The prefix function, Π

Following pseudocode computes the prefix fucnction, Π:

<u>Compute-Prefix-Function (p)</u>

```
m ← length[p]            //'p' pattern to be matched
Π[1] ← 0
k ← 0
    for q ← 2 to m
        do while k > 0 and p[k+1] != p[q]
    do k ← Π[k]
    End while
            If p[k+1] = p[q]
              then k ← k +1
    End if
            Π[q] ← k
    End for
  return Π
```

Example: compute Π for the pattern ‘p’ below:

P	a	b	a	b	a	c	a

Initially: m = length[p] = 7

Π[1] = 0

k = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0					

Step 1: q = 2, k=0

Π[2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1				

Step 2: q = 3, k = 0,

Π[3] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
Π	0	0	1	2			

Step 4: q = 5, k =2

Π[5] = 3

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3		

Step 5: q = 6, k = 3

Π[6] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	

Step 6: q = 7, k = 1

Π[7] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	1

After iterating 6 times, the prefix function computation is complete: →

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
Π	0	0	1	2	3	1	1

The KMP Matcher

The KMP Matcher, with pattern ‘p’, string ‘S’ and prefix function ‘Π’ as input, finds a match of p in S.
Following pseudocode computes the matching component of KMP algorithm:

KMP-Matcher(S,p)

```
1 n ← length[S]
2 m ← length[p]
3 Π ← Compute-Prefix-Function(p)
4 q ← 0                                  //number of characters matched
5 for i ← 1 to n                         //scan S from left to right
6    do while  q > 0 and p[q+1] != S[i]
7          do  q ← Π[q]                   //next character does not match
8     End while
8        if p[q+1] = S[i]
9          then q ← q + 1                //next character matches
10    End if
10       if q = m                        //is all of p matched?
11         then print "Pattern occurs with shift" i – m
12             q ← Π[ q]                 // look for the next match
13    End if
```

Note: KMP finds every occurrence of a ‘p’ in ‘S’. That is why KMP does not terminate in step 12, rather it searches remainder of ‘S’ for any more occurrences of ‘p’.

Illustration: given a String 'S' and pattern 'p' as follows:

S

b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

p

a	b	a	b	a	c	a

Let us execute the KMP algorithm to find whether 'p' occurs in 'S'.

For 'p' the prefix function, Π was computed previously and is as follows:

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
Π	0	0	1	2	3	1	1

Initially: n = size of S = 15;

m = size of p = 7

Step 1: i = 1, q = 0

comparing p[1] with S[1]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

P[1] does not match with S[1]. 'p' will be shifted one position to the right.

Step 2: i = 2, q = 0

comparing p[1] with S[2]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p		a	b	a	b	a	c	a

P[1] matches S[2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

Comparing p[2] with S[3]   p[2] does not match with S[3]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p		a	b	a	b	a	c	a							

Backtracking on p, comparing p[1] and S[3]

Step 4: i = 4, q = 0

comparing p[1] with S[4]   p[1] does not match with S[4]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p				a	b	a	b	a	c	a					

Step 5: i = 5, q = 0

comparing p[1] with S[5]   p[1] matches with S[5]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p					a	b	a	b	a	c	a				

Step 6: i = 6, q = 1

Comparing p[2] with S[6]   p[2] matches with S[6]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 7: i = 7, q = 2

Comparing p[3] with S[7]   p[3] matches with S[7]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 8: i = 8, q = 3

Comparing p[4] with S[8]   p[4] matches with S[8]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 9: i = 9, q = 4

Comparing p[5] with S[9]   p[5] matches with S[9]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p					a	b	a	b	a	c	a				

Step 10: i = 10, q = 5

Comparing p[6] with S[10]   p[6] doesn’t match with S[10]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p					a	b	a	b	a	c	a				

Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

Step 11: i = 11, q = 4

Comparing p[5] with S[11]   p[5] matches with S[11]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p							a	b	a	b	a	c	a		

Step 12: i = 12, q = 5

Comparing p[6] with S[12]

p[6] matches with S[12]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p							a	b	a	b	a	c	a		

Step 13: i = 13, q = 6

Comparing p[7] with S[13]

p[7] matches with S[13]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p							a	b	a	b	a	c	a		

Pattern ‘p’ has been found to completely occur in string ‘S’. The total number of shifts that took place for the match to be found are: i – m = 13 – 7 = 6 shifts.

Running - time analysis

- Compute-Prefix-Function (Π)

```
1  m ← length[p]        //'p' pattern to be matched
2  Π[1] ← 0
3  k ← 0
4      for q ← 2 to m
5          do while k > 0 and p[k+1] != p[q]
6          do k ← Π[k]
7             If p[k+1] = p[q]
8               then k ← k +1
9             Π[q] ← k
10     return Π
```

In the above pseudocode for computing the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step 1 to step 3 take constant time. Hence the running time of compute prefix function is Θ(m).

- KMP Matcher

```
1 n ← length[S]
2 m ← length[p]
3 Π ← Compute-Prefix-Function(p)
4 q ← 0
5 for i ← 1 to n
6   do while  q > 0 and p[q+1] != S[i]
7          do  q ← Π[q]
8      if p[q+1] = S[i]
9         then q ← q + 1
10     if q = m
11       then print "Pattern occurs with shift" i – m
12              q ← Π[ q]
```

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S'. Since step 1 to step 4 take constant time, the running time is dominated by this for loop. Thus running time of matching function is Θ(n).

Knuth-Morris-Pratt Algorithm

```
KMP-MATCHER(T, P)
n ← length[T]
m ← length[P]
π ← COMPUTE-PREFIX-FUNCTION(P)
q ← 0                                      # characters matched
for i ← 1 to n                             scan text left-to-right
    do while q > 0 and P[q + 1] ≠ T[i]
           do q ← π[q]                     next character does not match
       if P[q + 1] = T[i]
          then q ← q + 1                   next character matches
       if q = m                            Is all of P matched?
          then print "Pattern occurs with shift" i − m
               q ← π[q]                    Look for next match
```

$\Theta(m)$ in $\Theta(n)$ → line 3 (COMPUTE-PREFIX-FUNCTION)

using amortized analysis

$\Theta(n)$ (lines 5–12)

using amortized analysis

$\Theta(m+n)$ (whole algorithm)

Knuth-Morris-Pratt Algorithm

Worst Case

Amortized Analysis

Potential Method

$$\Phi(k) = k$$

k = current state of algorithm

Potential is never negative since π (k) >= 0 for all k

COMPUTE-PREFIX-FUNCTION(P)

```
m ← length[P]
π[1] ← 0
k ← 0                                  initial potential value
for q ← 2 to m
      do while k > 0 and P[k + 1] ≠ P[q]
              do k ← π[k]              potential decreases
         if P[k + 1] = P[q]
            then k ← k + 1             potential increases by <=1 in each execution of for loop body
         π[q] ← k
return π
```

Θ(m) in Θ(n)

amortized cost of loop body is in O(1)

Θ(m) loop iterations

Thank You !!!