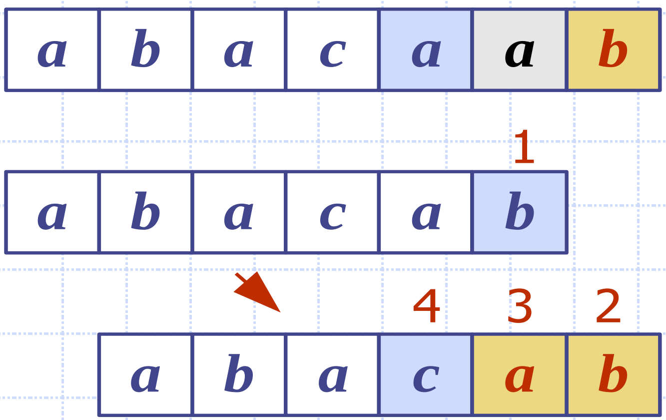
Pattern Matching



Introduction

- What is *string matching*?
  - Finding all occurrences of a *pattern* in a given *text* (or *body of text*)
- Many applications
  - While using editor/word processor/browser
  - Login name & password checking
  - Virus detection
  - Header analysis in data communications
  - DNA sequence analysis

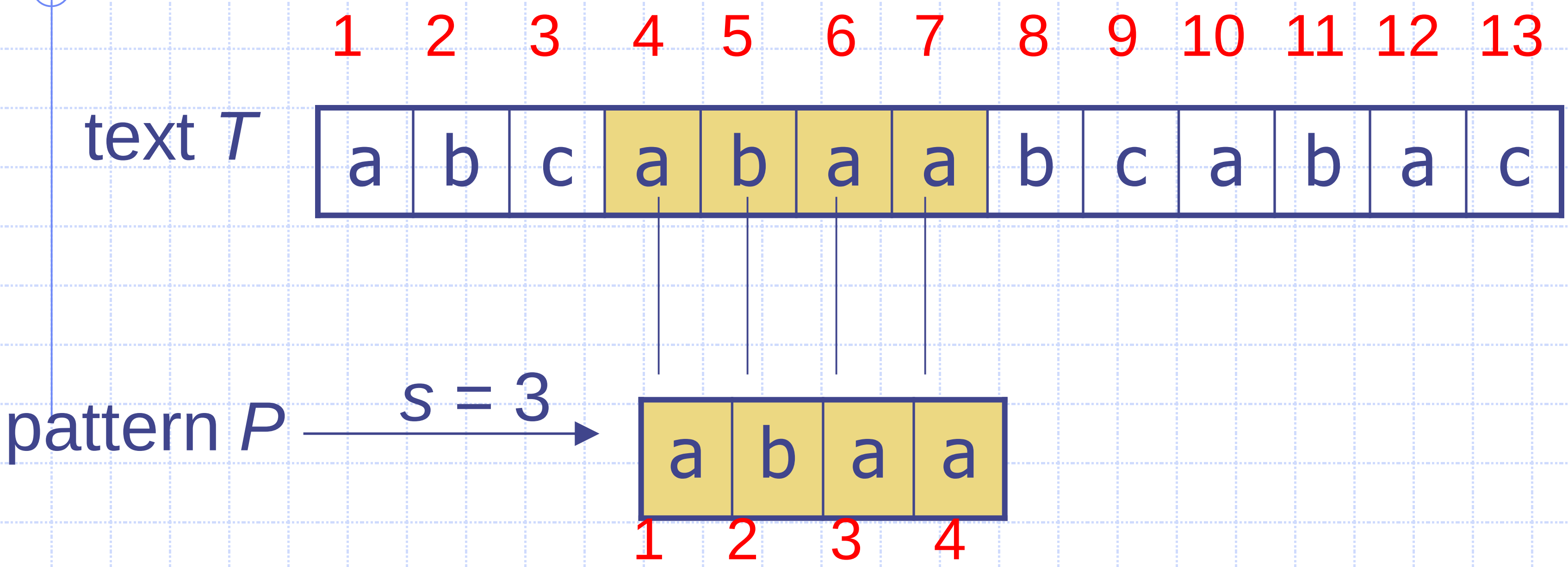
String-Matching Problem

- The *text* is in an array $T[1..n]$ of length n
- The *pattern* is in an array $P[1..m]$ of length m
- Elements of T and P are characters from a *finite alphabet* Σ
  - E.g., $\Sigma = \{0,1\}$ or $\Sigma = \{a, b, \ldots, z\}$
- Usually T and P are called *strings* of characters

String-Matching Problem ...contd

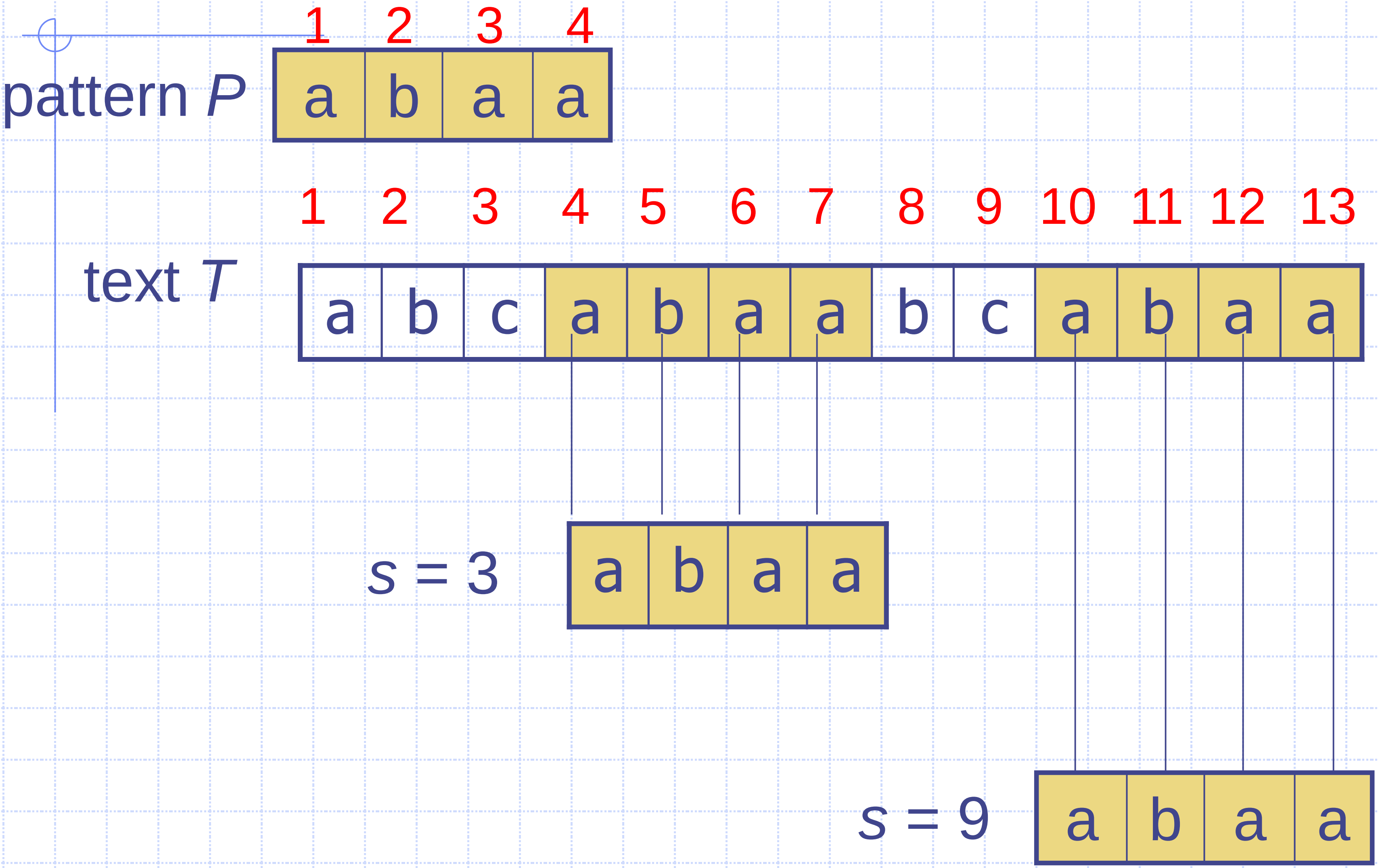
- We say that pattern *P* *occurs with shift s* in text *T* if:
  a) $0 \leq s \leq n\text{-}m$ and
  b) $T[(s+1)..(s+m)] = P[1..m]$
- If *P* occurs with shift *s* in *T*, then *s* is a *valid shift*, otherwise *s* is an *invalid shift*
- String-matching problem: finding all valid shifts for a given *T* and *P*

Example 1



shift s = 3 is a valid shift
(n=13, m=4 and $0 \le s \le n-m$ holds)

Example 2


1 2 3 4
pattern *P*
a b a a
1 2 3 4 5 6 7 8 9 10 11 12 13
text *T*
a b c a b a a b c a b a a
s = 3
a b a a
s = 9
a b a a

Terminology

- *Concatenation* of 2 strings x and y is xy
  - E.g., x="sri", y="lanka" $\Rightarrow$ xy = "srilanka"
- A string w is a *prefix* of a string x, if $x=wy$ for some string y
  - E.g., "srilan" is a prefix of "srilanka"
- A string w is a *suffix* of a string x, if $x=yw$ for some string y
  - E.g., "anka" is a suffix of "srilanka"

Naïve String-Matching Algorithm

Input: Text strings $T\,[1..n]$ and $P[1..m]$

Result: All valid shifts displayed

NAÏVE-STRING-MATCHER (T, P)

$n \leftarrow length[T]$

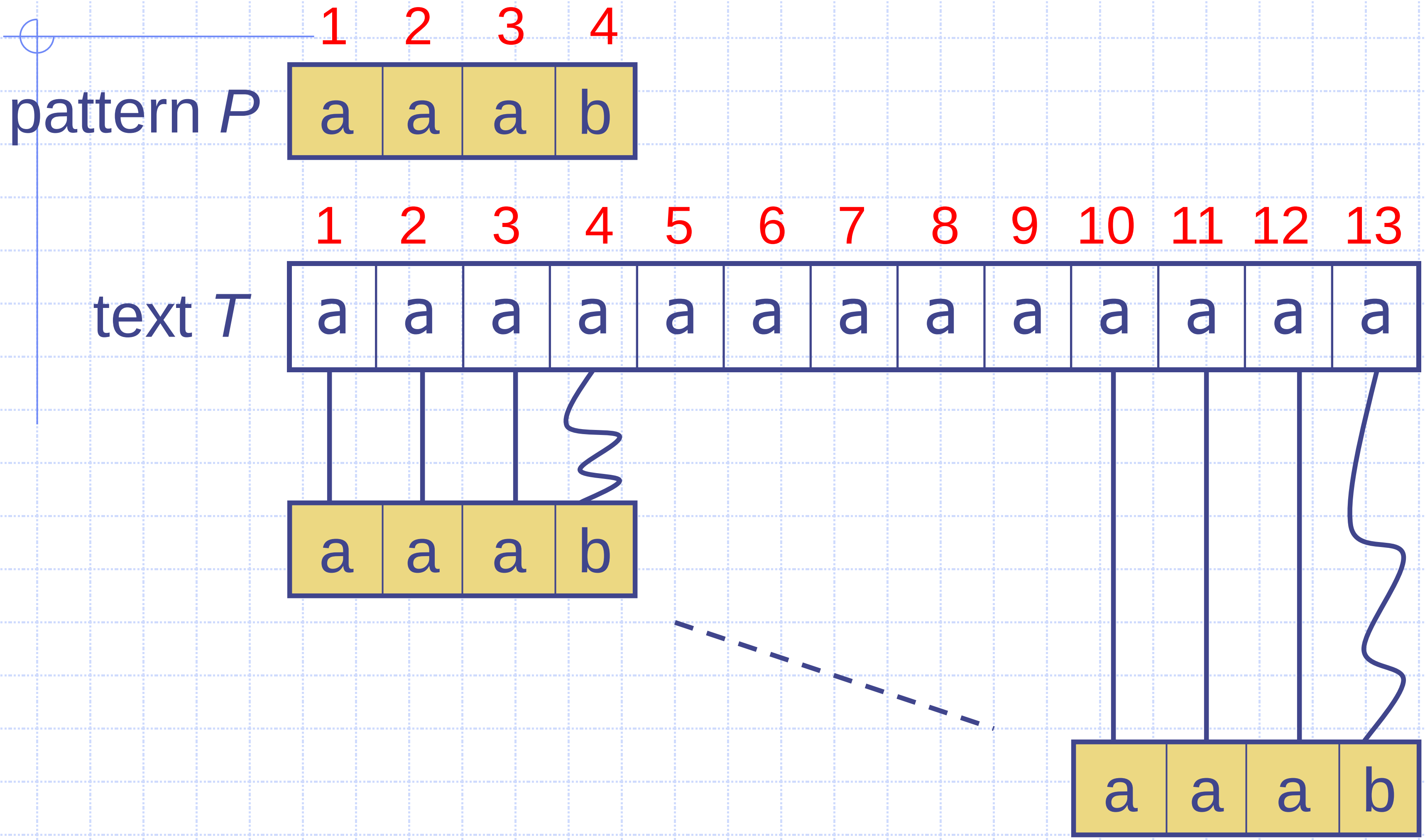
$m \leftarrow length[P]$

for $s \leftarrow 0$ **to** $n\text{-}m$

if $P[1..m] = T\,[(s+1)..(s+m)]$

print "pattern occurs with shift" s

Analysis: Worst-case Example



Worst-case Analysis

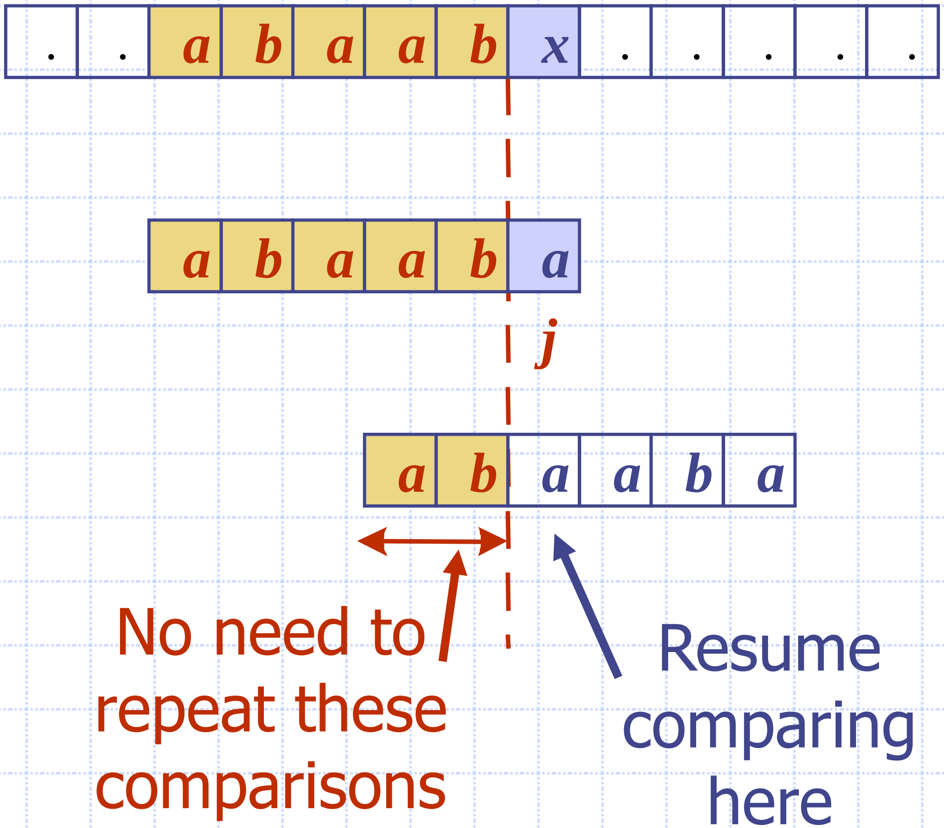
- There are m comparisons for each shift in the worst case
- There are n-m+1 shifts
- So, the worst-case running time is $\Theta((n-m+1)m)$
- Naïve method is inefficient because information from a shift is not used again

Other Approaches

- Naïve view is that it is always necessary to examine every character in T in order to locate a pattern P as a substring.
- This is not always the case…

The KMP Algorithm

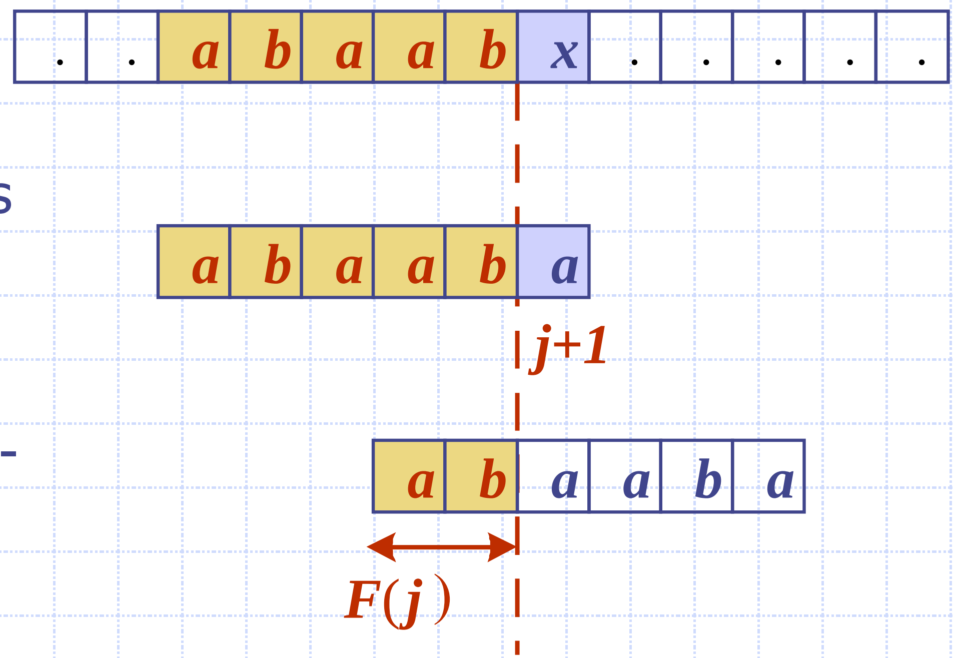
- Knuth-Morris-Pratt's algorithm compares the pattern to the text in **left-to-right**, but shifts the pattern more intelligently.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix of $P[1..j]$ that is a suffix of $P[2..j]$



KMP Prefix Function

- Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself
- The **Prefix function** $F(j)$ is defined as the size of the largest prefix of $P[1..j]$ that is also a suffix of $P[2..j]$
- Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j+1] \neq T[i]$ we set $j \leftarrow F(j)$

j	1	2	3	4	5	6
$P[j]$	*a*	*b*	*a*	*a*	*b*	*a*
$F(j)$	0	0	1	1	2	3



Components of KMP algorithm

- The prefix function, Π

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'.

- The KMP Matcher

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:

Compute-Prefix-Function (p)

```
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]
            If p[k+1] = p[q]
               then k ← k +1
            Π[q] ← k
    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

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					

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				

Step 3: q = 4, k = 1
Π[4] = 2

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

Step 4: $q = 5, k = 2$

$\Pi[5] = 3$

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

Step 5: $q = 6, k = 3$

$\Pi[6] = 0$

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

Step 6: $q = 7, k = 0$

$\Pi[7] = 1$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	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	0	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         if p[q+1] = S[i]
9            then q ← q + 1               //next character matches
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
```

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

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

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

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

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

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

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

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

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

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).

Complexity

- Using Potential method of amortized analysis
- Prefix Function's Complexity *O(m).*
- KMP's String matching alg's complexity: *O(n)*