Exact string matching

String vs. sequence

(Computer science vs. computational biology)

- String
  - a segment of *consecutive* characters.
  - usually called *sequence* in Biology.
- Sequence
  - need not be consecutive (in CS)
  - = String (in computational biology)
- Example:
  - *S* = “a t g a t g c a a t”
  - Substrings of *S*: “g a t g c”, “t g c a a t”.
  - Subsequences of *S*: “a g g t”, “a a a a”.

The exact string matching problem

- Input
  - a string P — the pattern
  - a string S — the text
- Output
  - all the occurrences of P in S.

Applications

- Computer Science
  - Dictionary, database
  - Search engines: Yahoo!, Google, …
- **Biology:**
  - **Biological sequence analysis**
- **Warm-up** for this course:
  - A well studied problem,
  - The idea/technique behind.

Notation for strings

- *S* is a string
  - |*S*| = the length of *S.*
  - substring: *S*[*i*…*j*].

A naïve algorithm

- Input: *S* and *P*.
- Output: all occurrences of *P* in *S*.

```
for i=1 to |S|
  if S[i…i+|P|-1] equals P
     output i;
```

Time complexity?

- $O(|S|^2\,|P|)$
- $O(|S|\,|P|)$
- $O(|P|\log|S|)$
- $O(|S| + |P|)$

Tightness of the time complexity

- $O(|S|\,|P|)$ is tight.
  - Why?
  - S = 1 1 1 1 1 1 1 1 1 1 1
  - P = 1 1 1 1 1 1 1
- $O(|S|^2\,|P|)$ is not tight.
  - Why?

Time complexity = $\Theta(|S|\,|P|)$

Knuth-Morris-Pratt

- [*SIAM J. Computing* 1977]
- Jumping as far as possible
  – far enough (for efficiency)
  – not too far (for correctness)

An example

```
S = x a b x y a b x y a b x z
P = a b x y a b x z
      a b x y a b x z
        a b x y a b x z
          a b x y a b x z
            a b x y a b x z
              a b x y a b x z
```

mismatch = a signal for “jumping”.

More, …

```
S = x a b x y a b x y a b x z
P = a b x y a b x z
      a b x y a b x z
        a b x y a b x z
          a b x y a b x z
            a b x y a b x z
              a b x y a b x z
```

What if there's no mismatch?

```
S = t t t t t t t t t
P = t t t t
      t t t t
        t t t t
          t t t t
            t t t t
              t t t t
```

Comments

- mismatch = a signal for “jumping”.
- Two types of jumps
  - Skipping several iterations
  - Skipping several comparisons in each iterations

For example, …

```
S = x a b x y a b x y a b x z
P = a b x y a b x z
      a b x y a b x z
        a b x y a b x z
          a b x y a b x z
            a b x y a b x z
              a b x y a b x z
```

Comments

- mismatch = a signal for "jumping".
- Two types of jumps
  - Skipping several iterations
  - Skipping several comparisons in each iterations
- How far is each jump?
  - Determinable merely from *P*.
  - That is, *S* is irrelevant.

For example, …

```
S = x a b x y a b x y a b x z
P = a b x y a b x z
      a b x y a b x z
        a b x y a b x z
          a b x y a b x z
            a b x y a b x z
              a b x y a b x z
```

Strategy

Preprocessing P in $O(|P|)$ time to obtain the necessary information that correctly guides the "jumps".

Components of KMP algorithm

- The prefix function, Π

The prefix function,Π(i) 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

- P_k: the first k characters of P (the k length prefix of P).
  - P = *ababddc*
  - P_1= *a*
  - P_3= *aba*
- Given $P[1,m]$, define the prefix-function:

$$\pi[j] = \max\{k : k < j \text{ and } P_k \sqsupset P_j\}$$

Suffix

Matching P with itself after (the smallest) shifting j-k!

The prefix function, Π

Following pseudo-code computes the prefix function, 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]
             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:

a	b	a	b	a	c	a

p

Initially: m = length[p] = 7
Π[1] = 0
k = 0

Step 1: q = 2, k=0
Π[2] = 0

k+1 q

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

k+1 q

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
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] = 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 = 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	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          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         print "Pattern occurs with shift" i – m
12         q ← Π[ q]                     // look for the next match
13    }
```

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	0	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							

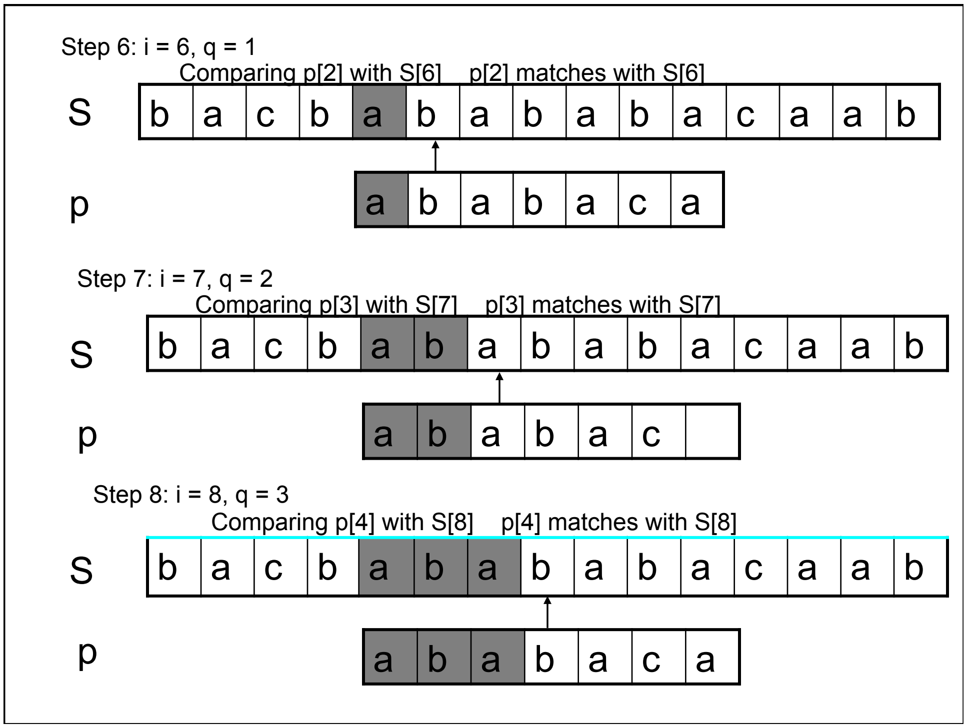
Backtracking on p, comparing p[1] and S[3]

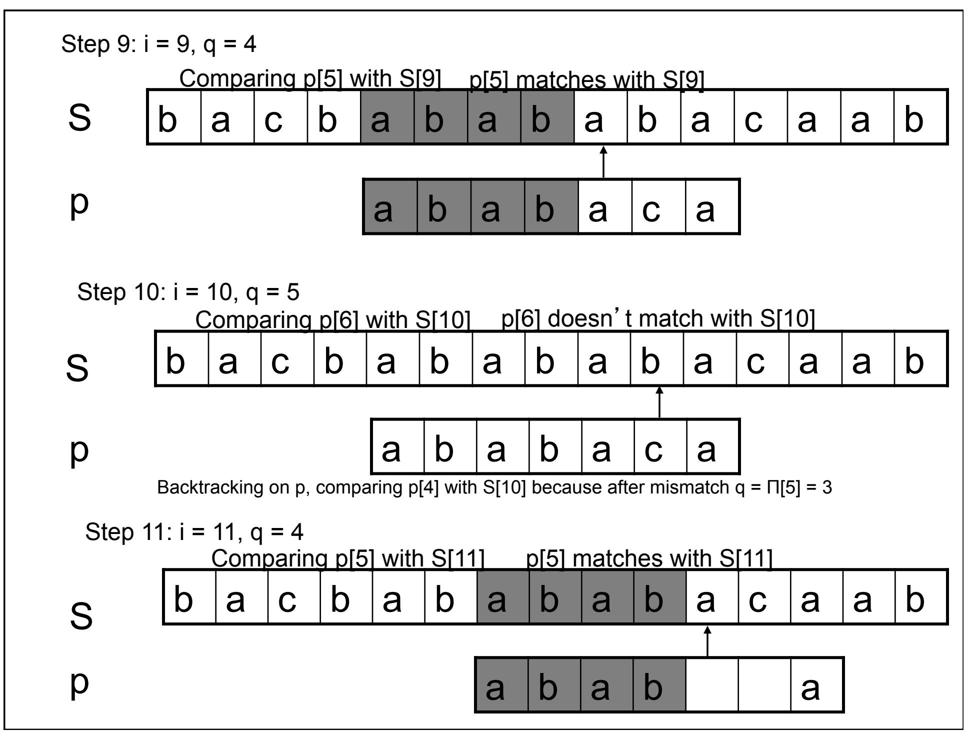
Step 4: i = 4, q = 0
comparing p[1] with S[4] p[1] does not match with S[4]

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
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

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 - complexity analysis

KMP Matcher

```
matched
Π[1] ← 0
 k ← 0
for q ← 2 to m
        do while k > 0 and p[k+1] !=
p[q]
              k ← Π[k]
        If p[k+1] = p[q]
               then k ← k +1
        Π[q] ← k
return Π

=Ω(|p|)
```

```
q ← 0
for i ← 1 to n
   do while  q > 0 and p[q+1] != S[i]
          q ← Π[q]
   if p[q+1] = S[i]
       then q ← q + 1
   if q = m        {
      print "Pattern occurs with shift" i –
  m
      q ← Π[ q]
   }

= Ω(|p|)
=O(|s|)
```