WIA2005: Algorithm Design and Analysis

Semester 2, Session 2016/17

Lecture 12: String Matching

Learning objectives

- Know string matching algorithm
  - Naïve algorithm
  - Rabin-Karp
  - Finite-automaton
  - Knuth-Morris-Pratt

Introduction

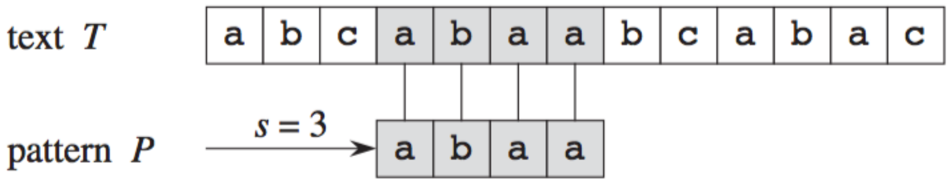
- Text-editing programs frequently need to find all occurrences of a pattern in the text.
- Typically, the text is a document being edited, and the pattern searched for is a particular word supplied by the user.
- Efficient algorithms for this problem—called "string matching"—can greatly aid the responsiveness of the text-editing program.
- Among their many other applications, string-matching algorithms search for particular patterns in DNA sequences.
- Internet search engines also use them to find Web pages relevant to queries.

String matching problem

- We formalize the string-matching problem as follows.
- We assume that the text is an array $T[1..n]$ of length n and that the pattern is an array $P[1..m]$ of length $m \le n$.
- We further assume that the elements of P and T are characters drawn from a finite alphabet Σ.
  - For example, we may have $\Sigma = \{0,1\}$ or $\Sigma = \{a, b, c, .., z\}$.
  - The character arrays P and T are often called ***strings*** of characters.

Naïve (Brute-force) string-matching algorithm

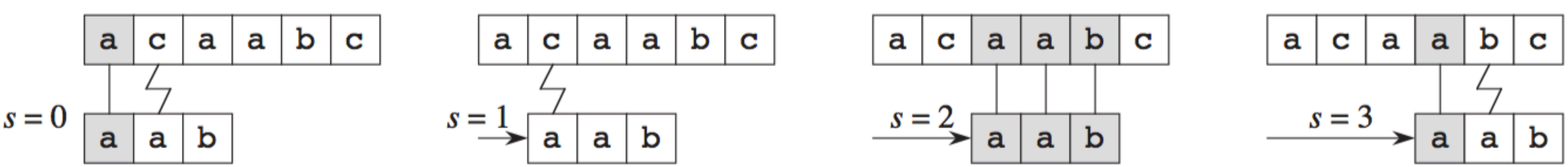
- An example of the string-matching problem, where we want to find all occurrences of the pattern P = abaa in the text T = abcabaabcabac.
- The pattern occurs only once in the text, at shift s = 3, which we call a valid shift.
- A vertical line connects each character of the pattern to its matching character in the text, and all matched characters are shaded.



Naive string-matching pseudocode

- The naive algorithm finds all valid shifts using a loop that checks the condition P = [1..m] = T[s + 1… s +m] for each of the n – m + 1 possible values of s.

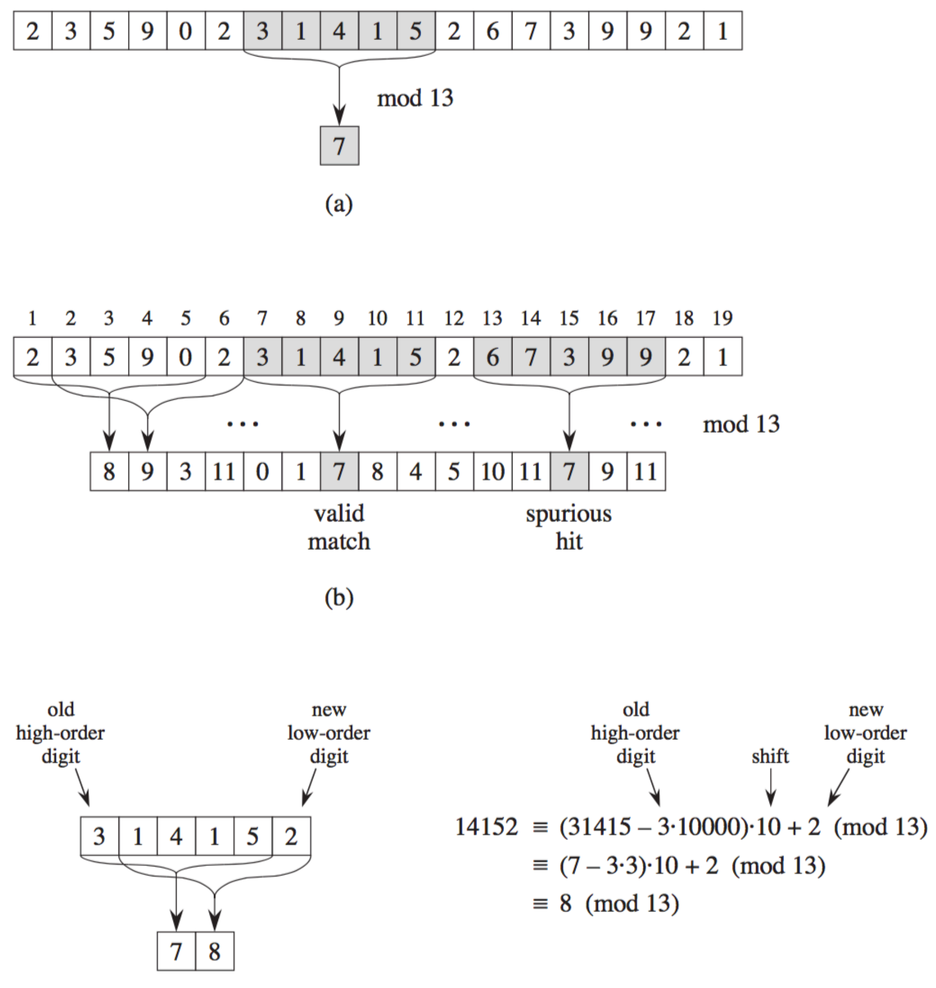
```
NAIVE-STRING-MATCHER(T, P)
n = T.length
m = P.length
for s = 0 to n − m
    if P[1..m] == T[s + 1..s + m]
        print "Pattern occurs with shift" s
```



(a) (b) (c) (d)

Rabin-Karp algorithm

- Rabin and Karp proposed a string-matching algorithm that performs well in practice and that also generalizes to other algorithms for related problems, such as two-dimensional pattern matching.
- Using hash (Rolling hash)



Rabin-Karp Pseudocode

```
RABIN-KARP-MATCHER(T, P, d, q)
n = T.length
m = P.length
h = d^(m-1) mod q
p = 0
t_0 = 0
for i = 1 to m                          // preprocessing
    p = (dp + P[i]) mod q
    t_0 = (dt_0 + T[i]) mod q
for s = 0 to n - m                      // matching
    if p == t_s
        if P[1..m] == T[s + 1..s + m]
            print "Pattern occurs with shift" s
    if s < n - m
        t_(s+1) = (d(t_s - T[s + 1]h) + T[s + m + 1]) mod q
```

String matching with finite automata

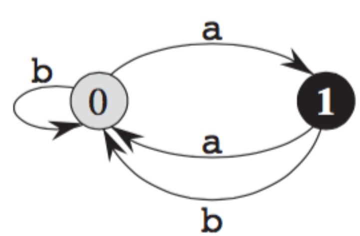
- Many string-matching algorithms build a finite automaton—a simple machine for processing information—that scans the text string T for all occurrences of the pattern P.
- These string-matching automata are very efficient: they examine each text character *exactly once*, taking constant time per text character.
    - especially for regular expressions used in compiler (parser)

Recap: Finite automata

- A ***finite automaton*** M, is a 5-tuple (Q, q_0, A, Σ, δ) where:
  - Q is a finite set of ***states***,
  - $q_0 \in Q$ is the ***start state***,
  - $A \subseteq Q$ is a distinguished set of ***accepting states***,
  - Σ is a finite ***input alphabet***,
  - δ is a function from $Q \times \Sigma$ into Q, called the ***transition function*** of M.

state	input a	input b
0	1	0
1	0	0

(a)

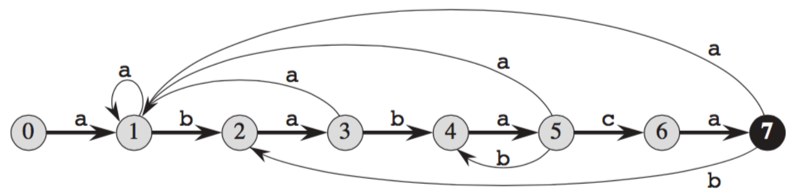


(b)

For example, on input abaaa, including the start state, this automaton enters the sequence of states <0, 1, 0, 1, 0, 1>, and so it accepts this input.
For input abbaa, it enters the sequence of states <0, 1, 0, 0, 1, 0>, and so it rejects this input.

String matching automata

- For a given pattern P , we construct a string-matching automaton in a preprocessing step before using it to search the text string.
- In order to specify the string-matching automaton corresponding to a given pattern P[1..m] we first define an auxiliary function σ, called the ***suffix function*** corresponding to P.



(a)

	input			
state	a	b	c	P
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	

(b)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

Recap: Prefix and Suffix

- Prefix
  - All characters in a string with one or more cut off the end.
  - Eg. A, Ay, Ayy, Ayyy are prefixes of Ayyyy

- Suffix
  - All characters in a string with one or more cut off in the beginning.
  - Eg. maoo, aoo, oo, o are suffixes of Lmaoo

Finite-automaton string-matching pseudocode

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
m = P.length
for q = 0 to m
    for each character a ∈ Σ
        k = min(m + 1, q + 2)
        repeat
            k = k − 1
        until P_k ⊐ P_q a
        δ(q, a) = k
return δ
```

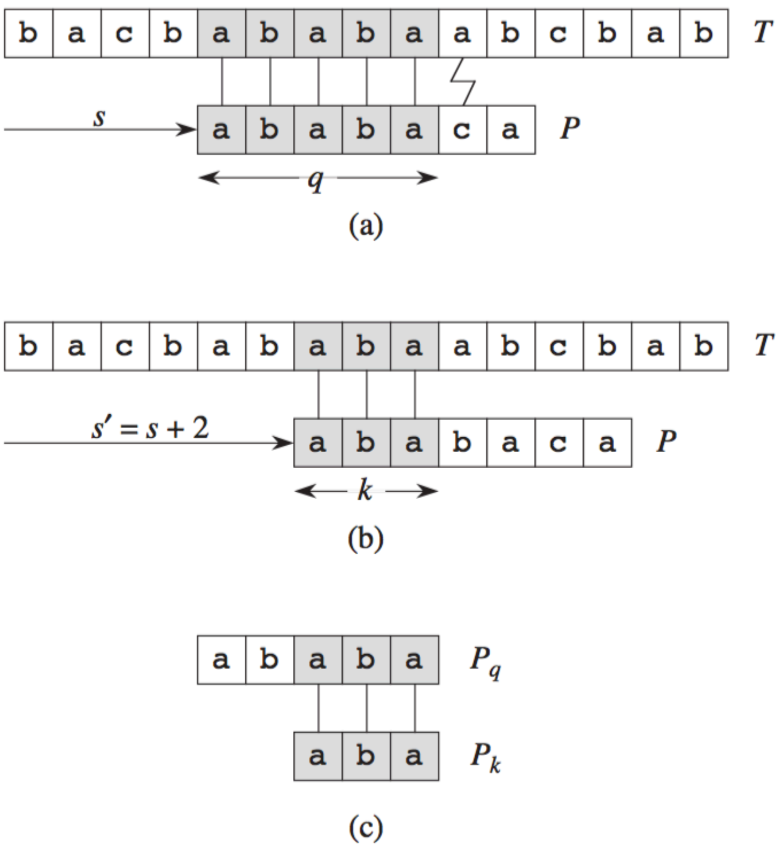
```
FINITE-AUTOMATON-MATCHER(T, δ, m)
n = T.length
q = 0
for i = 1 to n
    q = δ(q, T[i])
    if q == m
        print "Pattern occurs with shift" i − m
```

Knuth-Morris-Pratt (KMP) algorithm

- This algorithm avoids computing the transition function ı altogether, and its matching time is $O(n)$ using just an auxiliary function π, which we precompute from the pattern in time $O(m)$ and store in an array $\pi[1..m]$.
- The array allows us to compute the transition function δ efficiently (in an amortized sense) “on the fly” as needed.

Prefix function for a pattern

- The prefix function π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself.
- We can take advantage of this information to avoid testing useless shifts in the naive pattern-matching algorithm and to avoid precomputing the full transition function δ for a string-matching automaton.

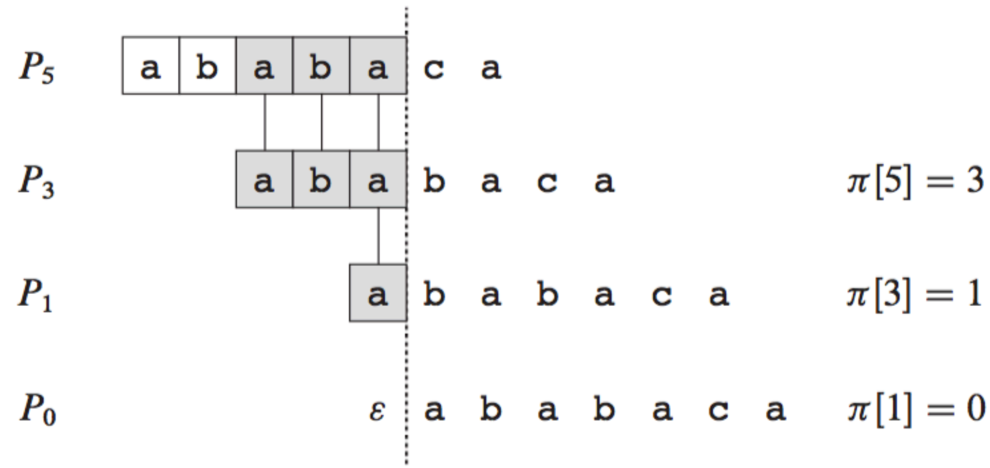


Prefix table

- Complete prefix function for the pattern ababaca:

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



(b)

KMP psudocode

- The pseudocode below gives the Knuth-Morris-Pratt matching algorithm as the procedure KMP-MATCHER.
- For the most part, the procedure follows from FINITE-AUTOMATON-MATCHER.
- KMP-MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION to compute

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

```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1 . . m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
        k = π[k]
    if P[k + 1] == P[q]
        k = k + 1
    π[q] = k
return π
```

Pre-processing and matching time

Algorithm	Preprocessing time	Matching time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite automaton	$O(m\,\lvert\Sigma\rvert)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

We are done. All the best!