

String Matching Algorithms

Unit 3

String Matching Problem

Motivations: text-editing, pattern matching in DNA sequences

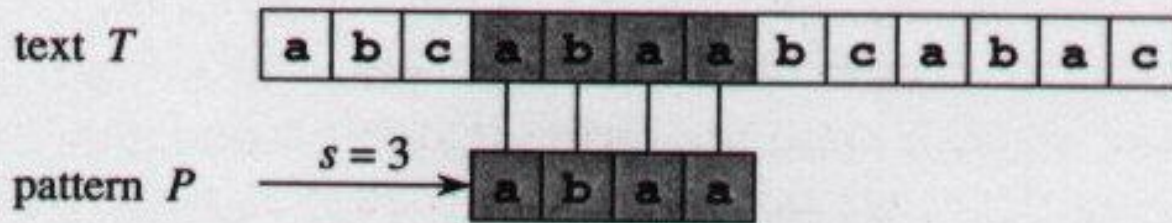


Figure 32.1 The string-matching problem. The goal is to find all occurrences of the pattern $P = abaa$ in the text $T = abcabaabcabac$. The pattern occurs only once in the text, at shift $s = 3$. The shift $s = 3$ is said to be a valid shift. Each character of the pattern is connected by a vertical line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

Text: array $T[1\dots n]$

$n > m$

Pattern: array $P[1\dots m]$

Array Element: Character from finite alphabet Σ

Pattern P occurs with shift s in T if $P[1\dots m] = T[s+1\dots s+m]$

$0 \leq s \leq n - m$

String Matching Algorithms

- Divide running time into preprocessing and matching time.
- Preprocessing: Setup some data structure based on pattern P.
- Matching: Perform actual matching by comparing characters from T with P and precomputed data structure.
- 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
- Finite Automaton-Based
 - Worst-case running time in $O(n + m |\Sigma|)$
- 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 : $w \boxed{x}$
- w is a suffix of x : $w \boxed{x}$
- prefix, suffix are *transitive*

ab abcca

cca abcca

Overlapping Suffix Lemma

Lemma 32.1 (*Overlapping-suffix lemma*)

Suppose that x , y , and z are strings such that $x \sqsupset z$ and $y \sqsupset z$. If $|x| \leq |y|$, then $x \sqsupset y$. If $|x| \geq |y|$, then $y \sqsupset x$. If $|x| = |y|$, then $x = y$.

Proof

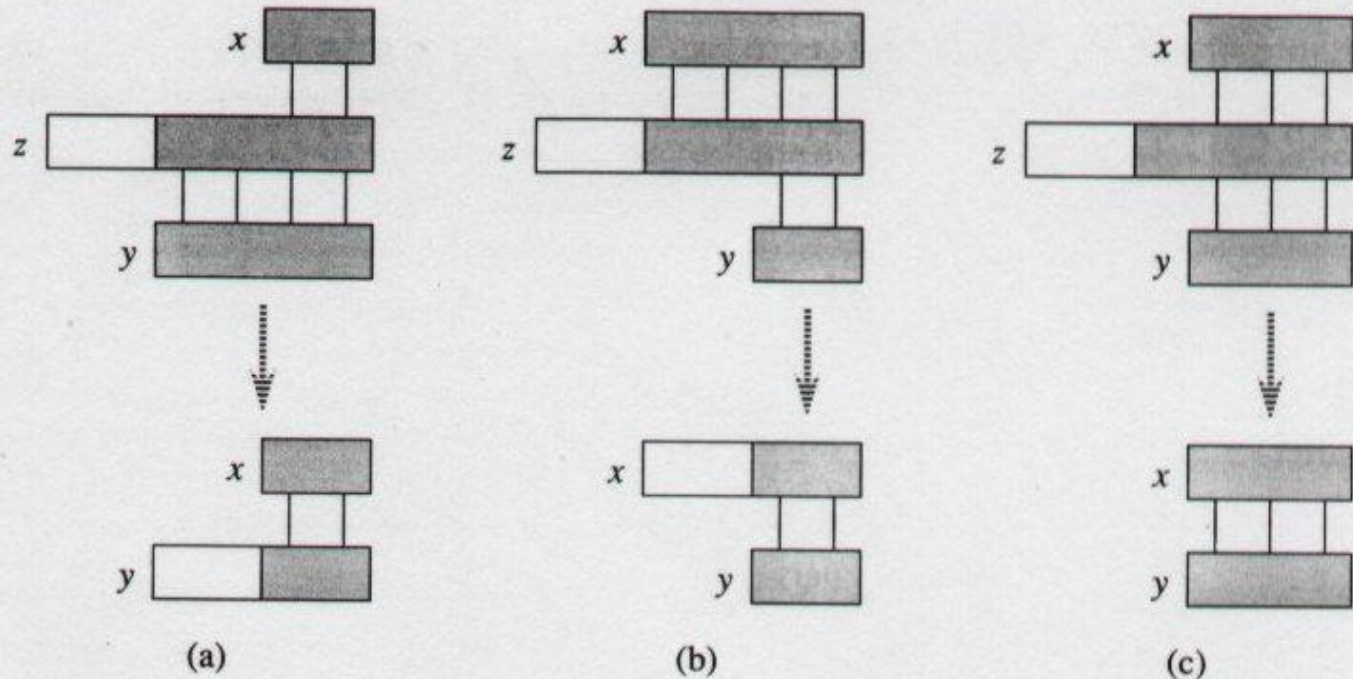


Figure 32.3 A graphical proof of Lemma 32.1. We suppose that $x \sqsupset z$ and $y \sqsupset z$. The three parts of the figure illustrate the three cases of the lemma. Vertical lines connect matching regions (shown shaded) of the strings. (a) If $|x| \leq |y|$, then $x \sqsupset y$. (b) If $|x| \geq |y|$, then $y \sqsupset x$. (c) If $|x| = |y|$, then $x = y$.

String Matching Algorithms

Naive Algorithm

Naive String Matching

NAIVE-STRING-MATCHER(T, P)

```
1  $n \leftarrow \text{length}[T]$ 
2  $m \leftarrow \text{length}[P]$ 
3 for  $s \leftarrow 0$  to  $n - m$ 
4   do if  $P[1..m] = T[s + 1..s + m]$ 
5     then print "Pattern occurs with shift"  $s$ 
```

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

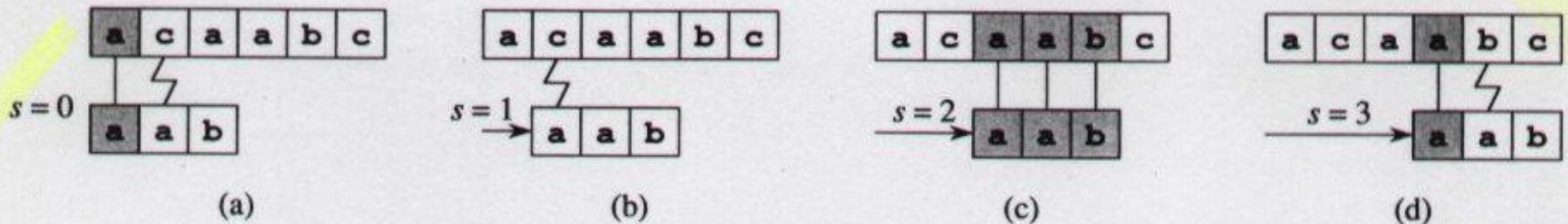


Figure 32.4 The operation of the naive string matcher for the pattern $P = aab$ and the text $T = acaabc$. We can imagine the pattern P as a “template” that we slide next to the text. Parts (a)–(d) show the four successive alignments tried by the naive string matcher. In each part, vertical lines connect 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).

String Matching Algorithms

Rabin-Karp

Rabin-Karp Algorithm

- Rabin-Karp string searching algorithm calculates a numerical (hash) value for the pattern p , and for each m -character substring of text t .
- Then it compares the numerical values instead of comparing the actual symbols.
- The algorithm slides the pattern, one by one, and matches the hash value of the substring of the text.
- If any match is found, it compares the pattern with the substring by naive approach.
- Otherwise it shifts to next substring of t to compare with p .
- The use of hashing converts the string to a numeric value which speeds up the process of matching.
- The algorithm exploits the fact that if two strings are equal then their hash values are also equal.
- Thus, the string matching is reduced to computing the hash value of the search pattern and then looking for substring with that hash value.

Rabin-Karp (1987)

- Consider (sub)strings as numbers. Characters in a string correspond to digits in a number written in radix-d notation (where $d = |\Sigma|$).

text: ab**cdef**ghijk

pattern: c d e f

$$h(\text{cdef}) = h(3, 4, 5, 6) = 18 \checkmark$$

$$h(\text{abcd}) = h(1, 2, 3, 4) = 10$$

$$h(\text{bcde}) = h(\text{abcd}) - 1 + 5 = 10 - 1 + 5 = 14$$

$$h(\text{cdef}) = h(\text{bcde}) - 2 + 6 = 14 - 2 + 6 = 18 \checkmark$$

| | |
|-----|----|
| a | 1 |
| b | 2 |
| c | 3 |
| d | 4 |
| e | 5 |
| f | 6 |
| g | 7 |
| h | 8 |
| i | 9 |
| j | 10 |
| k | 11 |
| ... | |

Rabin-Karp (1987)

Compute remaining t_i 's in $O(n-m)$ time

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

text: ab**cdef**ghijk

Fedc= 6543

pattern: c d e f

$$h(cdef) = h(3, 4, 5, 6) = 18 \checkmark$$

$$h(abcd) = h(1, 2, 3, 4) = 10$$

$$h(bcde) = h(abcd) - 1 + 5 = 10 - 1 + 5 = 14$$

$$h(cdef) = h(bcde) - 2 + 6 = 14 - 2 + 6 = 18 \checkmark$$

| | |
|-----|----|
| a | 1 |
| b | 2 |
| c | 3 |
| d | 4 |
| e | 5 |
| f | 6 |
| g | 7 |
| h | 8 |
| i | 9 |
| j | 10 |
| k | 11 |
| ... | |

Rabin-Karp

- Assume each character is digit in radix-d notation (e.g. $d=10$)
- p = decimal value of pattern

text: ab**cdef**ghijk multiplier: 10^{n-1} where n is the character position from the right

$h(abcd) = h(1, 2, 3, 4)$
 $= 1*10^3 + 2*10^2 + 3*10^1 + 4*10^0$
 $= 1000 + 200 + 30 + 4 = 1234$

$h(bcde) = (h(abcd) - 1*10^3) * 10 + 5$
 $= (1234 - 1000) * 10 + 5 = 2345$

$h(cdef) = (h(bcde) - 2*10^3) * 10 + 6$
 $= (2345 - 2000) * 10 + 6 = 3456$

| | |
|-----|----|
| a | 1 |
| b | 2 |
| c | 3 |
| d | 4 |
| e | 5 |
| f | 6 |
| g | 7 |
| h | 8 |
| i | 9 |
| j | 10 |
| k | 11 |
| ... | |

Compute remaining t_i 's in $O(n-m)$ time

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

We can signify the position of each char by multiplying by some constant raised to the power that corresponds (eg. 10^{n-1}) to its position.

Now, $H(1234) \neq H(4321)$ or any other permutations

Rabin-Karp

text: a**bcde**fg hijk

$$\begin{aligned}h(abcd) &= (1*10^3 + 2*10^2 + 3*10^1 + 4*10^0) \bmod 113 \\ &= (1234) \bmod 113 = 104\end{aligned}$$

$$\begin{aligned}h(bcde) &= (((h(abcd) - 1*10^3 \bmod 113)*10) \bmod 113 + 5) \bmod 113 \\ &= (((104 - 96)*10) \bmod 113 + 5) \bmod 113 \\ &= (80 \bmod 113 + 5) \bmod 113 = 85\end{aligned}$$

$$\text{Sanity check: } (2000 + 300 + 40 + 5) \bmod 113 = 85$$

If pattern was 1000 chars then we need to multiply by 10^9 which would be a huge number (integer overflow).

Therefore, divide with a prime number (eg 113 – now hash value will always be under/less than 113)

Rabin-Karp

text: abcdefghijk

$$\begin{aligned}h(\text{abcd}) &= h(1, 2, 3, 4) \\&= (1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0) \bmod 113 \\&= (1000 + 200 + 30 + 4) \bmod 113 \\&= (1234) \bmod 113 = 104\end{aligned}$$


$$\begin{aligned}h(\text{bcde}) &= (((h(\text{abcd}) - 1 \cdot 10^3 \bmod 113) \cdot 10) \bmod 113 + 5) \bmod 113 \\&= (((104 - 96) \cdot 10) \bmod 113 + 5) \bmod 113 \\&= (80 \bmod 113 + 5) \bmod 113 = 85\end{aligned}$$

$$\text{Sanity check: } (2000 + 300 + 40 + 5) \bmod 113 = 85$$

Example

- Example (1):
- Input: T = gtgatcagatcact, P = tca
- Output: Yes. gtgat**tc**agat**tc**act, shift=4, 9
-
- Example (2):
- Input: T = 189342670893, P = 1673
- Output: No.

Rabin-Karp Algorithm

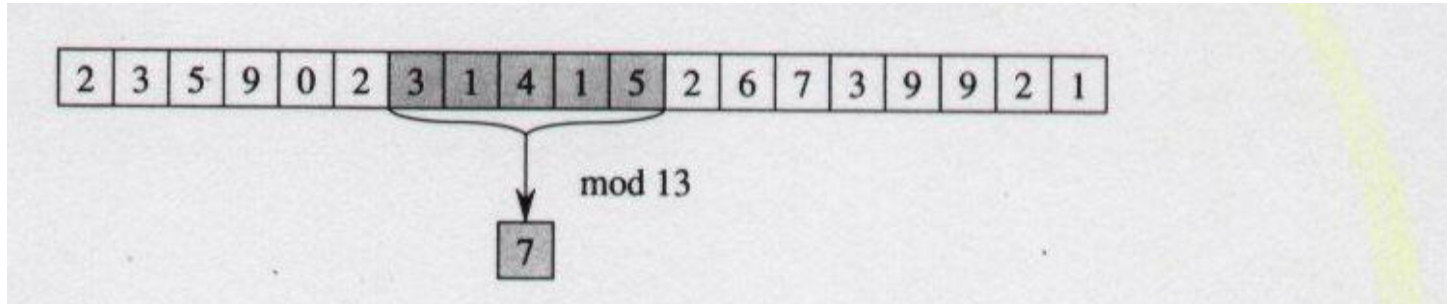
- Consider (sub)strings as numbers. Characters in a string correspond to digits in a number written in radix-d notation (where $d = |\Sigma|$).
- Assume each character is digit in radix-d notation (e.g. $d=10$)
- p = decimal value of pattern
- t_s = decimal value of substring $T[s+1..s+m]$ for $s = 0, 1, \dots, n-m$
- 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:
 - $p = P[m] + d(P[m-1] + d(P[m-2] + \dots + d(P[2] + dP[1])))$
- 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 scheme

- Consider a character as a number in a radix system, e.g., English alphabet as in radix-26.
- Pick up each m-length "number" starting from shift=0 through (n-m).
- So, T = gtgatcagatcact P=tga
in radix-4 (a/0, t/1, g/2, c/3) becomes
 - $gtg = '212'$ in base-4 $= 2*4^2 + 1*4^1 + 2*4^0 = 32 + 4 + 2$
 - $tga = '120'$ in base-4 $= 1*4^2 + 2*4^1 + 0*4^0 = 16 + 8 + 0$
- Then do the comparison with P - number-wise.
- (1) preprocess for (n-m) numbers on T and 1 for P,
- (2) compare the number for P with those computed on T.

Rabin-Karp scheme

- Problem: in case each number (p and t_s) is too large for comparison
- Solution: *Hash*, use modular arithmetic, with respect to a prime q .



- $31415 \% 13 = 7$
- New recurrence formula:
- $t_{s+1} = (d(t_s - h T[s+1]) + T[s+m+1]) \bmod q$,
- where $h = d^{m-1} \bmod q$.
- q is a prime number so that we do not get a 0 in the mod operation.
- The comparison is not perfect and may have spurious hit (see next slide).
- So, we need a naïve string matching when the comparison succeeds in modulo math.

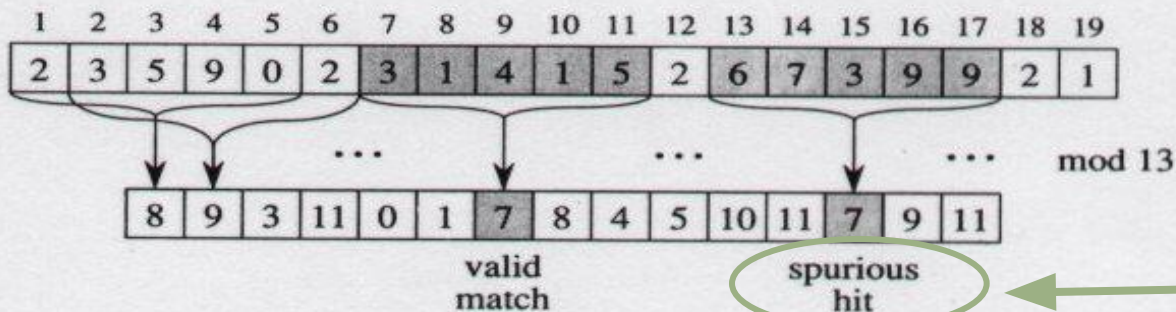
Rabin-Karp Algorithm (continued)

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



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

The comparison is not perfect and may have spurious hit (see example below). So, we need a naïve string matching when the comparison succeeds in modulo math.

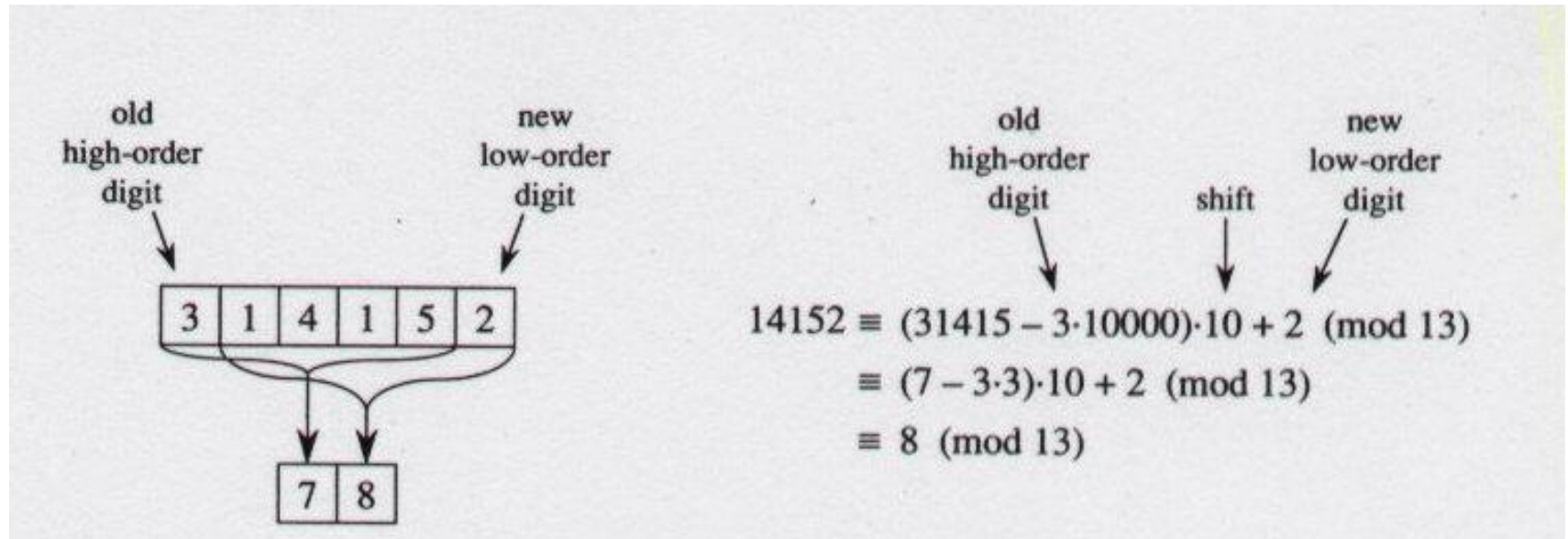


$$p = 31415$$

**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)



(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

- Compute p in $O(m)$ time using Horner's rule:
 - $p = P[m] + d(P[m-1] + d(P[m-2] + \dots + d(P[2] + dP[1])))$
- Compute t_0 similarly from $T[1..m]$ in $O(m)$ time
- Compute remaining t_i 's in $O(n-m)$ time
 - $t_{s+1} = d(t_s - d^{m-1}T[s+1]) + T[s+m+1]$
- Advantage: Calculating strings can reuse old results.
- Consider decimals: 43592.. and 43592..
- $3592 = (4359 - 4*1000)*10 + 2$
 $= (359)*10+2= 3590+2$
 $= 3592$
- General formula: $t_{s+1} = d(t_s - d^{m-1}T[s+1]) + T[s+m+1]$, in radix- d , where t_s is the corresponding number for the substring $T[s..(s+m)]$. Note, m is the size of P .

Rabin-Karp Algorithm:

Input: Text string T, Pattern string to search for P, radix to be used d ($= |\Sigma|$, for alphabet Σ), a prime q

Output: Each index over T where P is found

Rabin-Karp-Matcher (T, P, d, q)

n = length(T); m = length(P);

$h = d^{m-1} \bmod q$;

p = 0; $t_0 = 0$;

for i = 1 through m do // Preprocessing

 p = (d*p + P[i]) mod q;

$t_0 = (d * t_0 + T[i]) \bmod q$;

end for;

for s = 0 through (n-m) do // Matching

 if (p == t_s) then

 if (P[1..m] == T[s+1 .. s+m]) then //rule out spurious hit

 print the shift value as s;

 if (s < n-m) then

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

end for;

End algorithm.

Rabin-Karp Algorithm (continued)

RABIN-KARP-MATCHER(T, P, d, q)

d is radix q is modulus

1 $n \leftarrow \text{length}[T]$

2 $m \leftarrow \text{length}[P]$

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

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 $n - m$

Matching loop invariant: when line 10 executed

10 do if $p = t_s$

$t_s = T[s+1..s+m] \bmod q$

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

$\Theta(m)$

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

$\Theta(m)$

Try all possible shifts

What input generates worst case?

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

Rabin-Karp Algorithm (continued)

Worst Case

$\Theta(m)$ in $\Theta(n)$ →

$\Theta(m)$ {

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

Try all possible shifts

```

RABIN-KARP-MATCHER( $T, P, d, q$ )  d is radix  q is modulus
1   $n \leftarrow \text{length}[T]$ 
2   $m \leftarrow \text{length}[P]$ 
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      do  $t_0 \leftarrow (dt_0 + T[i]) \bmod q$ 
9  for  $s \leftarrow 0$  to  $n - m$   Matching loop invariant: when line 10 executed
10     do if  $p = t_s$    $t_s = T[s+1..s+m] \bmod q$ 
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  $t_{s+1} \leftarrow (d(t_s - T[s+1])h + T[s+m+1]) \bmod q$ 
    
```

Average Case

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

Estimate (chance that $t_s = p \bmod q$) = $1/q$



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 $q \geq m$



average-case running time is in $O(n+m)$

- <https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/>
- <https://www.youtube.com/watch?v=qQ8vS2btsxl>
- <https://www.youtube.com/watch?v=-ZeP4KHibkU>
- <https://www.youtube.com/watch?v=nxmaEFXKZa8>
- http://www.euroinformatica.ro/documentation/programming/!!!Algorithms_CORMEN!!!/DDU0213.html
- Hashing:
<https://www.youtube.com/watch?v=wWglAphfn2U&list=PLqM7alHXFySGwXaessYMemAnITqlZdZVE>

String Matching Algorithms

Finite Automata

Finite Automata

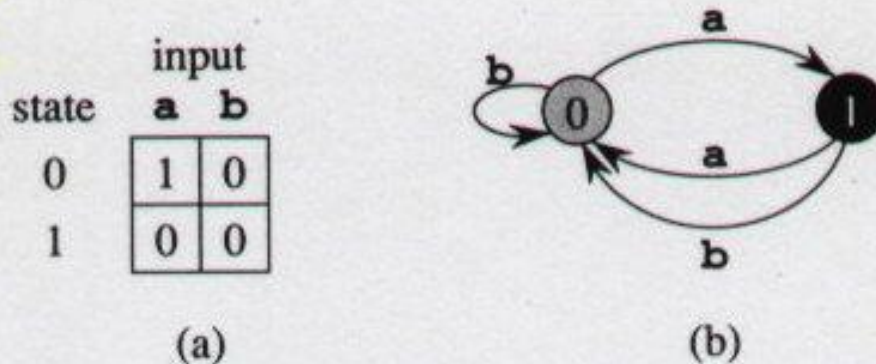


Figure 32.6 A simple two-state finite automaton with state set $Q = \{0, 1\}$, start state $q_0 = 0$, and input alphabet $\Sigma = \{a, b\}$. (a) A tabular representation of the transition function δ . (b) An equivalent state-transition diagram. State 1 is the only accepting state (shown blackened). Directed edges represent transitions. For example, the edge from state 1 to state 0 labeled b indicates $\delta(1, b) = 0$. This automaton accepts those strings that end in an odd number of a's. More precisely, a string x is accepted if and only if $x = yz$, where $y = \epsilon$ or y ends with a b, and $z = a^k$, where k is odd. For example, the sequence of states this automaton enters for input abaaa (including the start state) is $\langle 0, 1, 0, 1, 0, 1 \rangle$, and so it accepts this input. For input abbaa, the sequence of states is $\langle 0, 1, 0, 0, 1, 0 \rangle$, and so it rejects this input.

Strategy: Build automaton for pattern, then examine each text character once.

worst-case running time is in $\Theta(n) + \text{automaton creation time}$

Finite Automata

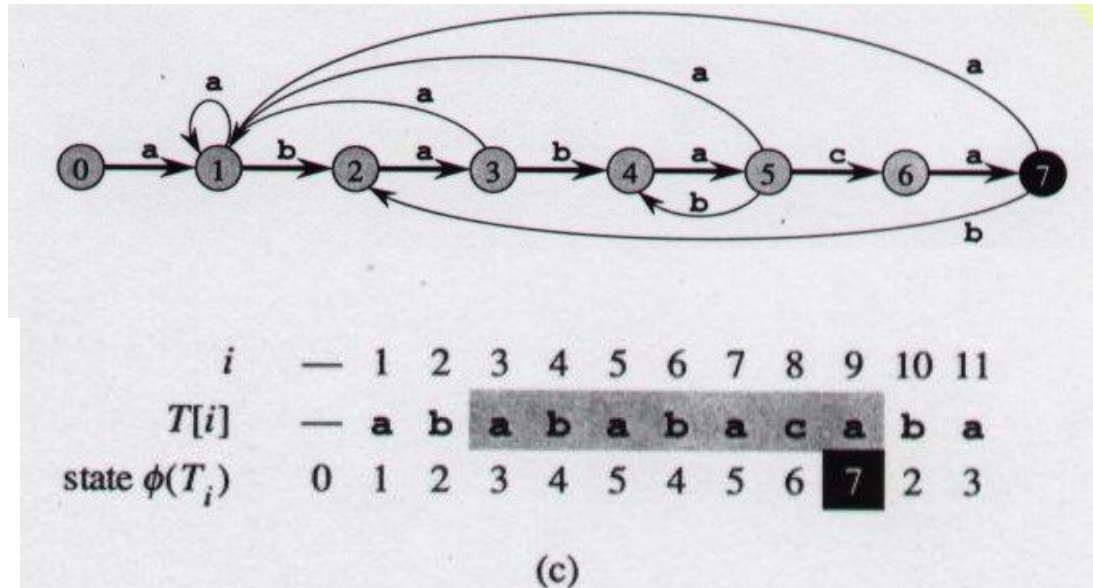
A *finite automaton* M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

- 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

A finite automaton M induces a function ϕ , called the *final-state function*, from Σ^* to Q such that $\phi(w)$ is the state M ends up in after scanning the string w . Thus, M accepts a string w if and only if $\phi(w) \in A$. The function ϕ is defined by the recursive relation

$$\begin{aligned}\phi(\varepsilon) &= q_0, \\ \phi(wa) &= \delta(\phi(w), a) \quad \text{for } w \in \Sigma^*, a \in \Sigma.\end{aligned}$$

String-Matching Automaton



Pattern = $P = \mathbf{ababaca}$

Automaton accepts strings **ending in P**

Figure 32.7 (a) A state-transition diagram for the string-matching automaton that accepts all strings ending in the string ababaca. State 0 is the start state, and state 7 (shown blackened) is the only accepting state. A directed edge from state i to state j labeled a represents $\delta(i, a) = j$. The right-going edges forming the “spine” of the automaton, shown heavy in the figure, correspond to successful matches between pattern and input characters. The left-going edges correspond to failing matches. Some edges corresponding to failing matches are not shown; by convention, if a state i has no outgoing edge labeled a for some $a \in \Sigma$, then $\delta(i, a) = 0$. (b) The corresponding transition function δ , and the pattern string $P = \mathbf{ababaca}$. The entries corresponding to successful matches between pattern and input characters are shown shaded. (c) The operation of the automaton on the text $T = \mathbf{abababacaba}$. Under each text character $T[i]$ is given the state $\phi(T_i)$ the automaton is in after processing the prefix T_i . One occurrence of the pattern is found, ending in position 9.

| 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)

| State | a | b | c | P |
|-------|---|---|---|---|
| 0 | 1 | | | a |
| 1 | | 2 | | b |
| 2 | 3 | | | a |
| 3 | | 4 | | b |
| 4 | 5 | | | a |
| 5 | | | 6 | c |
| 6 | 7 | | | a |
| 7 | | | | |

String-Matching Automaton

Suffix Function for P:

$\sigma(x)$ = length of longest prefix of P that is a suffix of x

$$\sigma(x) = \max \{k : P_k \sqsubseteq x\}$$

We define the string-matching automaton corresponding to a given pattern $P[1..m]$ as follows.

- The state set Q is $\{0, 1, \dots, m\}$. The start state q_0 is state 0, and state m is the only accepting state.
- The transition function δ is defined by the following equation, for any state q and character a :

$$\delta(q, a) = \sigma(P_q a) .$$

32.3

32.4

at each step: keeps track of longest pattern prefix that is a suffix of what has been read so far

String-Matching Automaton

Simulate behavior of string-matching automaton that finds occurrences of pattern P of length m in $T[1..n]$

FINITE-AUTOMATON-MATCHER(T, δ, m)

```
1   $n \leftarrow \text{length}[T]$ 
2   $q \leftarrow 0$ 
3  for  $i \leftarrow 1$  to  $n$ 
4      do  $q \leftarrow \delta(q, T[i])$ 
5          if  $q = m$ 
6              then  $s \leftarrow i - m$ 
7              print "Pattern occurs with shift"  $s$ 
```

**Worst
Case**

assuming automaton has **already been created...**

worst-case running time of **matching** is in $\Theta(n)$

String-Matching Automaton (continued)

The following procedure computes the transition function δ from a given pattern $P[1..m]$.

COMPUTE-TRANSITION-FUNCTION(P, Σ)

```
1  $m \leftarrow \text{length}[P]$ 
2 for  $q \leftarrow 0$  to  $m$ 
3   do for each character  $a \in \Sigma$ 
4     do  $k \leftarrow \min(m + 1, q + 2)$ 
5       repeat  $k \leftarrow k - 1$ 
6         until  $P_k \sqsupseteq P_q a$ 
7        $\delta(q, a) \leftarrow k$ 
8 return  $\delta$ 
```

source: 91.503 textbook Cormen et al.

**Worst
Case**

worst-case running time of **automaton creation** is in $O(m^2 |\Sigma|)$

can be improved to: $O(m |\Sigma|)$

worst-case running time of entire string-matching strategy
is in $O(m |\Sigma|) + O(n)$

automaton creation time

pattern matching time

String-Matching Automaton

Suffix Function for P:

$\sigma(x)$ = length of longest prefix of P that is a suffix of x

$$\sigma(x) = \max \{k : P_k \sqsubseteq x\}$$

We define the string-matching automaton corresponding to a given pattern $P[1..m]$ as follows.

- The state set Q is $\{0, 1, \dots, m\}$. The start state q_0 is state 0, and state m is the only accepting state.
- The transition function δ is defined by the following equation, for any state q and character a :

$$\delta(q, a) = \sigma(P_q a) .$$

32.3

$$\phi(T_i) = \sigma(T_i)$$

Automaton's operational invariant

32.4

at each step: keeps track of longest pattern prefix that is a suffix of what has been read so far

String-Matching Automaton (continued)

Correctness of matching procedure...

Lemma 32.2 (*Suffix-function inequality*)

For any string x and character a , we have $\sigma(xa) \leq \sigma(x) + 1$.

Proof Referring to Figure 32.8 let $r = \sigma(xa)$. If $r = 0$, then the conclusion $r \leq \sigma(x) + 1$ is trivially satisfied, by the nonnegativity of $\sigma(x)$. So assume that $r > 0$. Now, $P_r \sqsupset xa$, by the definition of σ . Thus, $P_{r-1} \sqsupset x$, by dropping the a from the end of P_r and from the end of xa . Therefore, $r - 1 \leq \sigma(x)$, since $\sigma(x)$ is largest k such that $P_k \sqsupset x$. ■

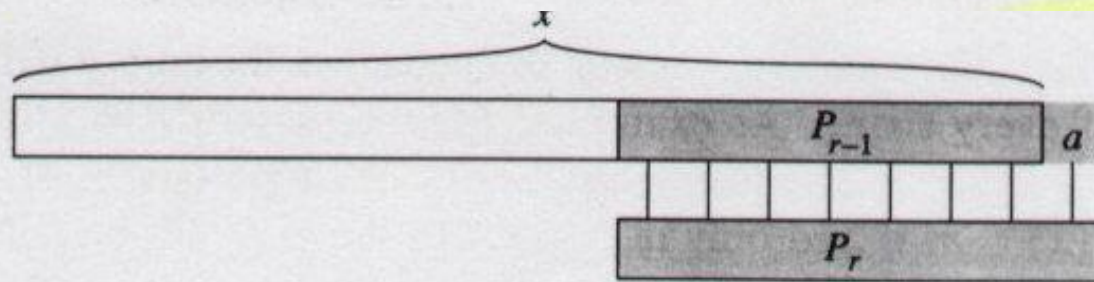


Figure 32.8 An illustration for the proof of Lemma 32.2. The figure shows that $r \leq \sigma(x) + 1$, where $r = \sigma(xa)$.

String-Matching Automaton (continued)

Correctness of matching procedure...

Lemma 32.3 (*Suffix-function recursion lemma*)

For any string x and character a , if $q = \sigma(x)$, then $\sigma(xa) = \sigma(P_q a)$.

Proof From the definition of σ , we have $P_q \sqsupset x$. As Figure 32.9 shows, $P_q a \sqsupset xa$. If we let $r = \sigma(xa)$, then $r \leq q + 1$ by Lemma 32.2. Since $P_q a \sqsupset xa$, $P_r \sqsupset xa$, and $|P_r| \leq |P_q a|$, Lemma 32.1 implies that $P_r \sqsupset P_q a$. Therefore, $r \leq \sigma(P_q a)$, that is, $\sigma(xa) \leq \sigma(P_q a)$. But we also have $\sigma(P_q a) \leq \sigma(xa)$, since $P_q a \sqsupset xa$. Thus, $\sigma(xa) = \sigma(P_q a)$. ■

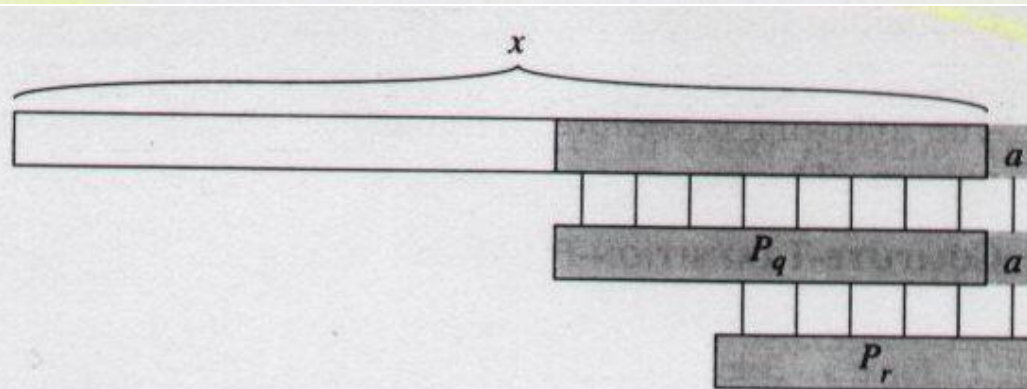


Figure 32.9 An illustration for the proof of Lemma 32.3 The figure shows that $r = \sigma(P_q a)$, where $q = \sigma(x)$ and $r = \sigma(xa)$. source: 91.503 textbook Cormen et al.

String-Matching Automaton (continued)

Correctness of matching procedure...

Theorem 32.4

If ϕ is the final-state function of a string-matching automaton for a given pattern P and $T[1..n]$ is an input text for the automaton, then

$$\phi(T_i) = \sigma(T_i)$$

for $i = 0, 1, \dots, n$.

Proof The proof is by induction on i . For $i = 0$, the theorem is trivially true, since $T_0 = \varepsilon$. Thus, $\phi(T_0) = \sigma(T_0) = 0$.

Now, we assume that $\phi(T_i) = \sigma(T_i)$ and prove that $\phi(T_{i+1}) = \sigma(T_{i+1})$. Let q denote $\phi(T_i)$, and let a denote $T[i+1]$. Then,

$$\begin{aligned}\phi(T_{i+1}) &= \phi(T_i a) && \text{(by the definitions of } T_{i+1} \text{ and } a) \\ &= \delta(\phi(T_i), a) && \text{(by the definition of } \phi) \\ &= \delta(q, a) && \text{(by the definition of } q) \\ &= \sigma(P_q a) && \text{(by the definition 32.3 of } \delta) \\ &= \sigma(T_i a) && \text{(by Lemma 32.3 and induction)} \\ &= \sigma(T_{i+1}) && \text{(by the definition of } T_{i+1}).\end{aligned}$$

By induction, the theorem is proved.

source: 91.503 textbook Cormen et al. ■

String-Matching Automaton (continued)

The following procedure computes the transition function δ from a given pattern $P[1..m]$.

COMPUTE-TRANSITION-FUNCTION(P, Σ)

```
1  $m \leftarrow \text{length}[P]$ 
2 for  $q \leftarrow 0$  to  $m$ 
3   do for each character  $a \in \Sigma$ 
4     do  $k \leftarrow \min(m + 1, q + 2)$ 
5       repeat  $k \leftarrow k - 1$ 
6         until  $P_k \sqsupseteq P_q a$ 
7        $\delta(q, a) \leftarrow k$ 
8 return  $\delta$ 
```

source: 91.503 textbook Cormen et al.

**Worst
Case**

worst-case running time of **automaton creation** is in $O(m^2 |\Sigma|)$

can be improved to: $O(m |\Sigma|)$

worst-case running time of entire string-matching strategy
is in $O(m |\Sigma|) + O(n)$

automaton creation time

pattern matching time

- <https://www.youtube.com/watch?v=uhMFMBpKih4>
- <https://www.youtube.com/watch?v=-ZeP4KHibkU&t=183s>
- http://www.euroinformatica.ro/documentation/programming/!!!Algorithms_CORMEN!!!/DDU0213.html

String Matching Algorithms

Horspool's Algorithm

Horspool's Algorithm

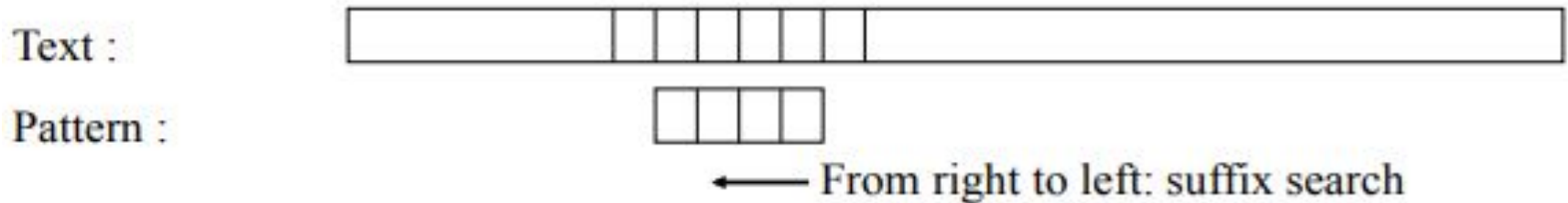
- It is possible in some cases to search text of length n in less than n comparisons!
- Horspool's algorithm is a relatively simple technique that achieves this distinction for many (but not all) input patterns. The idea is to perform the comparison from right to left instead of left to right.
- Here we don't shift the pattern by one character, but instead shift as large as possible.

Horspool's Algorithm

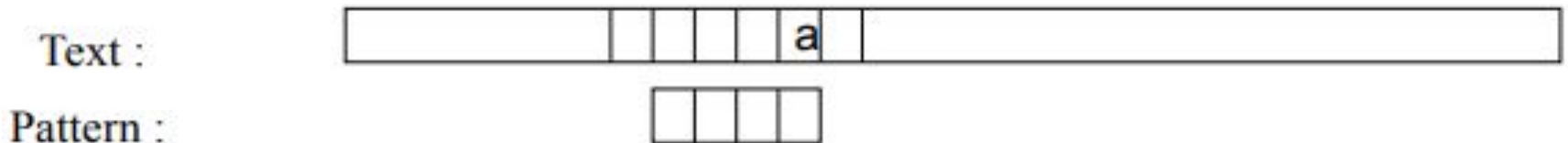
- However, if such an algorithm had to check all the characters of the pattern on every trial, it would lose much of this superiority.
- Fortunately, the idea of **input enhancement** makes repetitive comparisons unnecessary.
- We can **precompute shift sizes** and store them in a table.
- The table will be indexed by all possible characters that can be encountered in a text, including, for natural language texts, the space, punctuation symbols, and other special characters.
- If there is a character mismatch, how far can we shift the pattern, with no possibility of missing the first match within the text?

Horspool's Algorithm

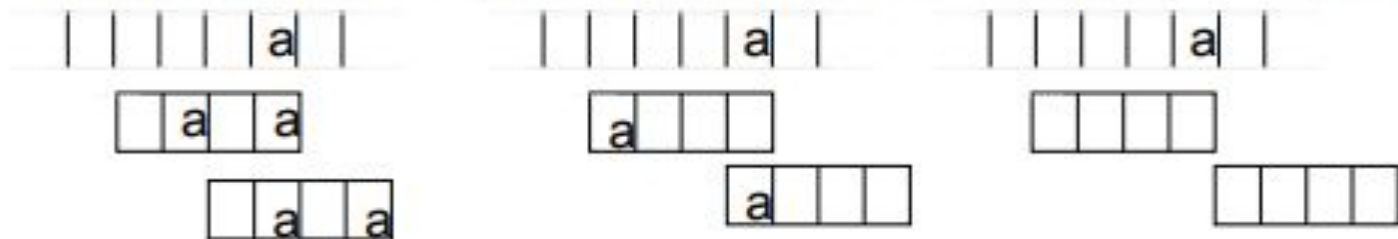
- How the comparison is made?



- Which is the next position of the window?



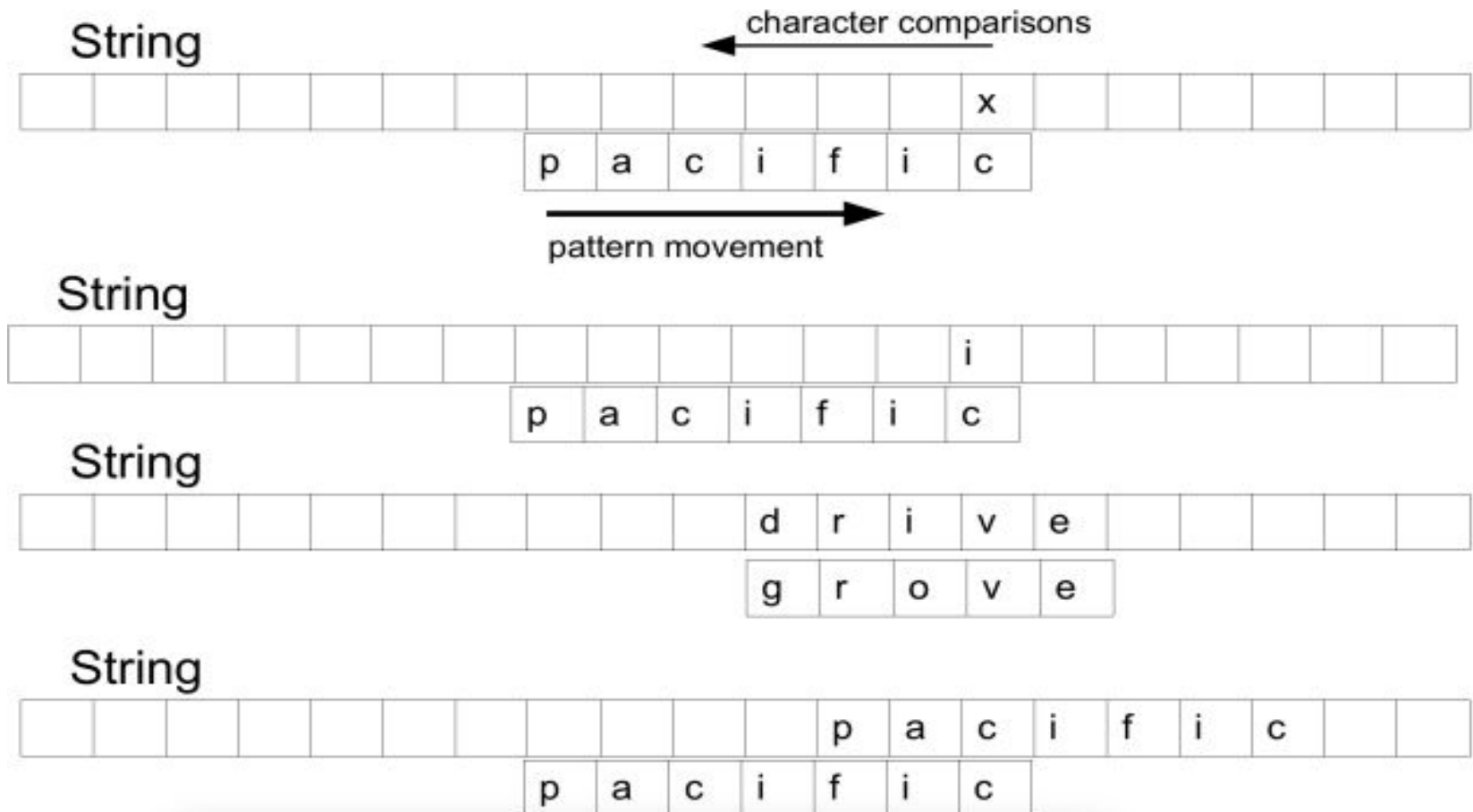
It depends of where appears the last letter of the text, say it 'a', in the pattern



Then it is necessary a preprocess that determines the length of the shift.

Horspool's Algorithm

- Four possibilities:



Horspool's algorithm

- If there is a character mismatch, **how far can we shift** the pattern, with no possibility of missing the first match within the text?

- What if the last character in the pattern is compared with a character in the text that does not occur in the pattern at all?

Text: ... ABCDEFG ...
Pattern: BOUTELL

- Look at first (rightmost) character in the part of the text that is compared to the pattern:

- The character is not in the pattern

.....**C**..... { C not in pattern}
BAOBAB

- The character is in the pattern (but not the rightmost)

.....**O**..... (O occurs once in pattern)
BAOBAB

.....**A**..... (A occurs twice in pattern)
BAOBAB

- The rightmost characters do match

.....**B**.....
BAOBAB

Horspool's Algorithm

- Four possibilities:

Case 1 If there are no c 's in the pattern—e.g., c is letter S in our example—we can safely shift the pattern by its entire length (if we shift less, some character of the pattern would be aligned against the text's character c that is known not to be in the pattern):

```

s0    ...    S                                ...    sn-1
          //
        B A R B E R
              B A R B E R
```

Case 2 If there are occurrences of character c in the pattern but it is not the last one there—e.g., c is letter B in our example—the shift should align the rightmost occurrence of c in the pattern with the c in the text:

```

s0    ...    B                                ...    sn-1
          //
        B A R B E R
              B A R B E R
```


Horspool's Algorithm

- Four possibilities:

Case 3 If c happens to be the last character in the pattern but there are no c 's among its other $m - 1$ characters—e.g., c is letter R in our example—the situation is similar to that of Case 1 and the pattern should be shifted by the entire pattern's length m :

```

s0    ...           M E R           ...    sn-1
                        X || ||
                L E A D E R
                        L E A D E R
```

Case 4 Finally, if c happens to be the last character in the pattern and there are other c 's among its first $m - 1$ characters—e.g., c is letter R in our example—the situation is similar to that of Case 2 and the rightmost occurrence of c among the first $m - 1$ characters in the pattern should be aligned with the text's c :

```

s0    ...           A R           ...    sn-1
                        X ||
                R E O R D E R
                        R E O R D E R
```

These examples clearly demonstrate that right-to-left character comparisons can lead to farther shifts of the pattern than the shifts by only one position

Horspool's Algorithm

- The table's entries will indicate the shift sizes computed by the formula:

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m - 1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m - 1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases}$$

- Initialize all the entries to the pattern's length m and**
- scan the pattern left to right repeating the following step $m - 1$ times: for the j th character of the pattern ($0 \leq j \leq m - 2$), **overwrite its entry in the table with $m - 1 - j$** , which is the character's distance to the last character of the pattern.*
- Note that since the algorithm scans the pattern from left to right, the last overwrite will happen for the character's rightmost occurrence—exactly as we would like it to be.

Horspool's Algorithm

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| B | A | R | B | E | R |

- $(m - 1 - j)$ i.e. the length of the pattern- actual index -1
- A : $6 - 1 - 1 = 4$ B : $6 - 1 - 0 = 5$ B : $6 - 1 - 3 = 2$ (overwrite the previous B value)
- R : $6 - 1 - 2 = 3$ E : $6 - 1 - 4 = 1$

| | | | | |
|---|---|---|---|---|
| A | B | E | R | * |
| 4 | 2 | 1 | 3 | 6 |

ALGORITHM *ShiftTable*($P[0..m - 1]$)

//Fills the shift table used by Horspool's and Boyer-Moore algorithms

//Input: Pattern $P[0..m - 1]$ and an alphabet of possible characters

//Output: $Table[0..size - 1]$ indexed by the alphabet's characters and

// filled with shift sizes computed by formula (7.1)

for $i \leftarrow 0$ to $size - 1$ do $Table[i] \leftarrow m$

for $j \leftarrow 0$ to $m - 2$ do $Table[P[j]] \leftarrow m - 1 - j$

return $Table$

Horspool's Algorithm

Horspool's algorithm

- Step 1** For a given pattern of length m and the alphabet used in both the pattern and text, construct the shift table as described above.
- Step 2** Align the pattern against the beginning of the text.
- Step 3** Repeat the following until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all m characters are matched (then stop) or a mismatching pair is encountered. In the latter case, retrieve the entry $t(c)$ from the c 's column of the shift table where c is the text's character currently aligned against the last character of the pattern, and shift the pattern by $t(c)$ characters to the right along the text.

Horspool's Algorithm

ALGORITHM *HorspoolMatching*($P[0..m-1]$, $T[0..n-1]$)

//Implements Horspool's algorithm for string matching
//Input: Pattern $P[0..m-1]$ and text $T[0..n-1]$
//Output: The index of the left end of the first matching substring
// or -1 if there are no matches

ShiftTable($P[0..m-1]$) //generate *Table* of shifts
 $i \leftarrow m-1$ //position of the pattern's right end

while $i \leq n-1$ **do**

$k \leftarrow 0$ //number of matched characters

while $k \leq m-1$ **and** $P[m-1-k] = T[i-k]$ **do**

$k \leftarrow k+1$

if $k = m$ //

return $i - m + 1$

else $i \leftarrow i + \text{Table}[T[i]]$

return -1

Horspool's Algorithm

EXAMPLE As an example of a complete application of Horspool's algorithm, consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

| character c | A | B | C | D | E | F | ... | R | ... | Z | _ |
|---------------|---|---|---|---|---|---|-----|---|-----|---|---|
| shift $t(c)$ | 4 | 2 | 6 | 6 | 1 | 6 | 6 | 3 | 6 | 6 | 6 |

| A | B | E | R | * |
|---|---|---|---|---|
| 4 | 2 | 1 | 3 | 6 |

The actual search in a particular text proceeds as follows:

```

J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R           B A R B E R
      B A R B E R       B A R B E R
            B A R B E R           B A R B E R

```



In running only make 12 comparisons, less than the length of the text! (24 chars)

Horspool's Algorithm

- Worst-case efficiency of Horspool's algorithm is in $O(nm)$.
- *But for* random texts, it is in $\theta(n)$, *Horspool's* algorithm is obviously faster on average than the brute-force algorithm.

| | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | - |
| 1 | 2 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 3 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |

BARD LOVED BANANAS

BAOBAB

BAOBAB

BAOBAB

BAOBAB (unsuccessful search)


```
pattern = abracadabra
text =
abracadabtabradabracadabcbadaxbrabbracadabraxxxxxabracadabracadabra
shiftTable:  a3 b2 r1 a3 c6 a3 d4 a3 b2 r1 a3 x11
abracadabtabradabracadabcbadaxbrabbracadabraxxxxxabracadabracadabra
abracadabra
abracadabtabradabracadabcbadaxbrabbracadabraxxxxxabracadabracadabra
    abracadabra
abracadabtabradabracadabcbadaxbrabbracadabraxxxxxabracadabracadabra
    abracadabra
abracadabtabradabracadabcbadaxbrabbracadabraxxxxxabracadabracadabra
        abracadabra
abracadabtabradabracadabcbadaxbrabbracadabraxxxxxabracadabracadabra
            abracadabra
abracadabtabradabracadabcbadaxbrabbracadabraxxxxxabracadabracadabra
                abracadabra
```

```
pattern = abracadabra
text =
abracadabtabradabracadabcbadaxbrabbracadabraxxxxxabracadabracadabra
shiftTable:  a3 b2 r1 a3 c6 a3 d4 a3 b2 r1 a3 x11
```

```
abracadabtabradabracadabcbadaxbrabbracadabraxxxxxabracadabracadabra
      abracadabra
abracadabtabradabracadabcbadaxbrabbracadabraxxxxxabracadabracadabra
      abracadabra
abracadabtabradabracadabcbadaxbrabbracadabraxxxxxabracadabracadabra
      abracadabra
abracadabtabradabracadabcbadaxbrabbracadabraxxxxxabracadabracadabra
      abracadabra
abracadabtabradabracadabcbadaxbrabbracadabraxxxxxabracadabracadabra
      abracadabra
abracadabtabradabracadabcbadaxbrabbracadabraxxxxxabracadabracadabra
      abracadabra
```

Using brute force, we would have to compare the pattern to 50 different positions in the text before we find it; with Horspool, only 13 positions are tried.

- String:
 - G T A C T A G A G G A C G T A T G T A C T G
- Pattern:
 - A T G T A
- Generate the shift table
- Show the steps of the algorithm

Boyer Moore

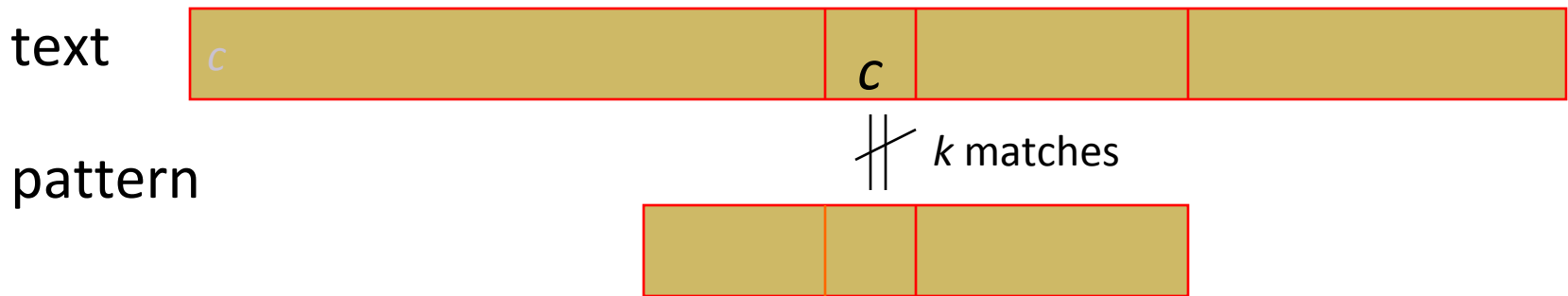
- Similar idea to Horspool's algorithm in that comparisons are made right to left, but is more sophisticated in how to shift the pattern.
- When determining how far to shift after a mismatch, Horspool only uses the text character corresponding to the rightmost pattern character.
- Often there is a partial match (from the right) before a mismatch occurs.
- Boyer-Moore takes into account k , the number of matched characters (from the right) before a mismatch occurs.
- If $k=0$, we do the same shift as Horspool's algorithm.

Boyer Moore

- Based on two main ideas:
 - compare pattern characters to text characters from right to left
 - precompute the shift amounts in two tables
 - **bad-symbol table** indicates how much to shift based on the text's character that causes a mismatch
 - **good-suffix table** indicates how much to shift based on matched part (suffix) of the pattern

Bad-symbol shift in Boyer-Moore algorithm

- If the rightmost character of the pattern doesn't match, BM algorithm acts as Horspool's
- If the rightmost character of the pattern does match, BM compares preceding characters right to left until either all pattern's characters match or a mismatch on text's character c is encountered after $k > 0$ matches



bad-symbol shift $d_1 = \max\{t(c) - k, 1\}$
where $t(c)$ is the value from the Horspool shift table

Bad-symbol shift in Boyer-Moore algorithm

For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter S in the text, we can shift the pattern by $t_1(S) - 2 = 6 - 2 = 4$ positions:

| | | | | | | |
|-------|-----|---|---|---|-----|-----------|
| s_0 | ... | S | E | R | ... | s_{n-1} |
| | | X | | | | |
| | | B | A | R | B | E |
| | | | | | B | A |
| | | | | | R | B |
| | | | | | E | R |

The same formula can also be used when the mismatching character c of the text occurs in the pattern, provided $t_1(c) - k > 0$. For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter A, we can shift the pattern by $t_1(A) - 2 = 4 - 2 = 2$ positions:

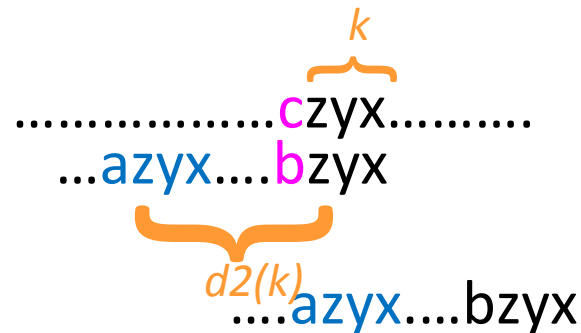
| | | | | | | |
|-------|-----|---|---|---|-----|-----------|
| s_0 | ... | A | E | R | ... | s_{n-1} |
| | | X | | | | |
| | | B | A | R | B | E |
| | | | | | B | A |
| | | | | | R | B |
| | | | | | E | R |

To summarize, the bad-symbol shift d_1 is computed by the Boyer-Moore algorithm either as $t_1(c) - k$ if this quantity is positive and as 1 if it is negative or zero. This can be expressed by the following compact formula:

$$d_1 = \max\{t_1(c) - k, 1\}. \quad (7.2)$$

Good-suffix shift in Boyer-Moore algorithm

- Good-suffix shift d_2 is applied after $0 < k < m$ last characters were matched
- $d_2(k)$ = the distance between (the last letter of) the matched suffix of size k and (the last letter of) its rightmost occurrence in the pattern that is not preceded by the same character preceding the suffix



| k | pattern | d_2 |
|-----|-----------------|-------|
| 1 | ABC <u>B</u> AB | 2 |
| 2 | AB <u>CB</u> AB | 4 |

Good-suffix shift in Boyer-Moore algorithm

- Good-suffix shift d_2 is applied after $0 < k < m$ last characters were matched
- $d_2(k)$ = the distance between (the last letter of) the matched suffix of size k and (the last letter of) its rightmost occurrence in the pattern that is not preceded by the same character preceding the suffix
- If there is no such occurrence, match the longest part (tail) of the k -character suffix with corresponding prefix;
if there are no such suffix-prefix matches, $d_2(k) = m$

Boyer-Moore Algorithm

After matching successfully $0 < k < m$ characters, the algorithm shifts the pattern right by

$$d = \max \{d_1, d_2\}$$

where $d_1 = \max\{t(c) - k, 1\}$ is bad-symbol shift

$d_2(k)$ is good-suffix shift

Boyer-Moore Algorithm

EXAMPLE As a complete example, let us consider searching for the pattern BAOBAB in a text made of English letters and spaces. The bad-symbol table looks as follows:

| | | | | | | | | | |
|----------|---|---|---|---|-----|---|-----|---|---|
| c | A | B | C | D | ... | 0 | ... | Z | _ |
| $t_1(c)$ | 1 | 2 | 6 | 6 | 6 | 3 | 6 | 6 | 6 |

The good-suffix table is filled as follows:

| k | pattern | d_2 |
|-----|----------------------------------|-------|
| 1 | BAO <u>B</u> A <u>B</u> | 2 |
| 2 | <u>B</u> AOBAB | 5 |
| 3 | <u>B</u> A <u>O</u> B <u>A</u> B | 5 |
| 4 | <u>B</u> A <u>O</u> B <u>A</u> B | 5 |
| 5 | <u>B</u> A <u>O</u> B <u>A</u> B | 5 |

Boyer-Moore Algorithm

B E S S _ K N E W _ A B O U T _ B A O B A B S
 B A O B A B

$$d_1 = t_1(K) - 0 = 6$$

B A O B A B

$$d_1 = t_1(_) - 2 = 4$$

B A O B A B

$$d_2 = 5$$

$$d_1 = t_1(_) - 1 = 5$$

$$d = \max\{4, 5\} = 5$$

$$d_2 = 2$$

$$d = \max\{5, 2\} = 5$$

B A O B A B

| | | | | | | | | | |
|----------|---|---|---|---|-----|---|-----|---|---|
| c | A | B | C | D | ... | O | ... | Z | _ |
| $t_1(c)$ | 1 | 2 | 6 | 6 | 6 | 3 | 6 | 6 | 6 |

The good-suffix table is filled as follows:

| k | pattern | d_2 |
|-----|---------------|-------|
| 1 | <u>BAOBAB</u> | 2 |
| 2 | <u>BAOBAB</u> | 5 |
| 3 | <u>BAOBAB</u> | 5 |
| 4 | <u>BAOBAB</u> | 5 |
| 5 | <u>BAOBAB</u> | 5 |

The Boyer-Moore algorithm

- Step 1** For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.
- Step 2** Using the pattern, construct the good-suffix shift table as described earlier.
- Step 3** Align the pattern against the beginning of the text.
- Step 4** Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after $k \geq 0$ character pairs are matched successfully. In the latter case, retrieve the entry $t_1(c)$ from the c 's column of the bad-symbol table where c is the text's mismatched character. If $k > 0$, also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the

number of positions computed by the formula

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

- Prof. Karen Daniels
- <https://www.youtube.com/watch?v=4Xyhb72LCX4>
- <https://www.youtube.com/watch?v=IkL6RkQvpMM>
- <https://www.youtube.com/watch?v=PHXAOKQk2dw&t=335s>