Design and Analysis of Algorithms

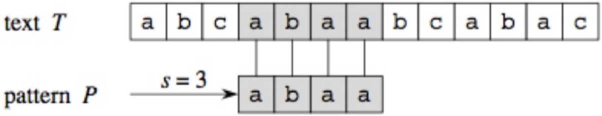
Mohammad GANJTABESH

mgtabesh@ut.ac.ir

School of Mathematics, Statistics and Computer Science,
University of Tehran,
Tehran, Iran.

Exact String Matching

Finding all occurrences of a pattern in a text.



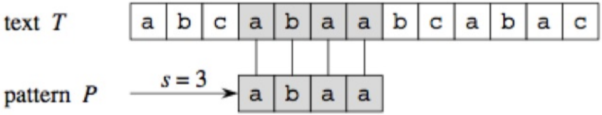
- Native Algorithm (Brute Force)
- Rabin-Karp
- Finite State Automata
- Knuth-Morris-Pratt (KMP)

Notaion and Problem Definition

- Σ: a given alphabet
- T: an string over Σ^n $(T[1 \cdots n])$
- P: an string over Σ^m $(P[1 \cdots m])$
- ϵ: empty string of length 0
- xy: concatenation of strings x and y
- $w \sqsubset x$: w is a prefix of x
- $w \sqsupset x$: w is a suffix of x
- T_k: the prefix $T[1 \cdots k]$ of T
- P_k: the prefix $P[1 \cdots k]$ of P
- $T_0 = P_0 = \epsilon$

Definition

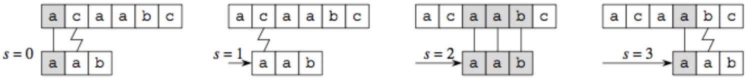
A shift s is valid iff $0 \leqslant s \leqslant n - m$ and $P[1 \cdots m] = T[s+1 \cdots s+m]$.
String matching problem: find all valid shifts.



Native Algorithm (Brute Force)

- Match the pattern string against the input string character by character.
- When there is a mismatch, shift the whole pattern string right by one character and start again at the beginning.

```
NAIVE-STRING-MATCHER(T, P)
m ← length[T]
n ← 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"
```



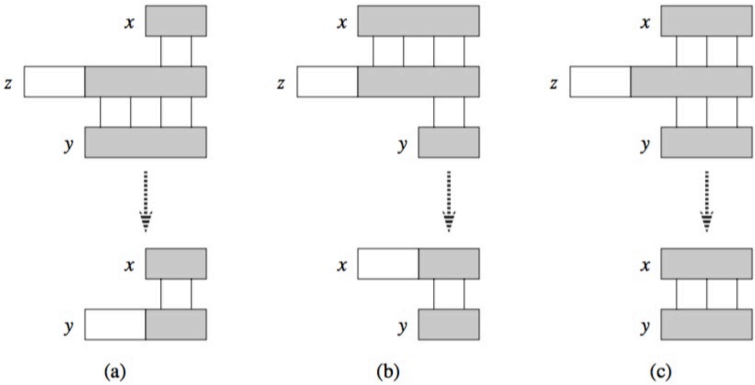
(a) (b) (c) (d)

Time Complexity: $\Theta((n - m + 1) \times m)$ (Consider $T = a^n$ and $P = a^m$).

String Matching property

Lemma

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



Rabin-Karp Algorithm

- Performs well in practice and can be used in two-dimensional pattern matching.
- Uses elementary number-theoretic notions (the equivalence of two numbers modulo a third number).
- Assume that each character is a digit in radix-d notation, where $d = |\Sigma|$.
- A string of length k can be seen as a length-k number.

Rabin-Karp Algorithm

- Let p denotes the corresponding decimal value of pattern $P[1 \cdots m]$.
- Similarly, t_s denotes the decimal value of length-m substring $T[s+1 \cdots s+m]$, for $s = 0, 1, \cdots, n-m$.
- Certainly, $t_s = p$ iff $T[s+1 \cdots s+m] = P[1 \cdots m]$; thus, s is a valid shift iff $t_s = p$.
- If we could compute p in time $\Theta(m)$ and all the t_s values in a total of $\Theta(n-m+1)$ time, then we could determine all valid shifts s in time $\Theta(m) + \Theta(n-m+1) = \Theta(n)$ by comparing p with each of the t_s's.

Rabin-Karp Algorithm

- We can compute p in time $\Theta(m)$ using Horners rule:

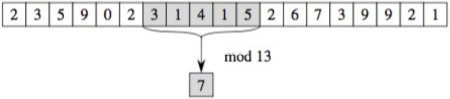
  $$p = P[m]+d(P[m-1]+d(P[m-2]+\cdots+d(P[2]+dP[1])\cdots)).$$

- The value t_0 can be similarly computed from $T[1\cdots m]$ in time $\Theta(m)$.
- To compute the remaining values $t_1, t_2, \cdots, t_{n-m}$ in time $\Theta(n-m)$, it suffices to observe that t_{s+1} can be computed from t_s in constant time, since

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

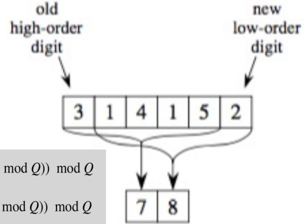
Rabin-Karp Algorithm

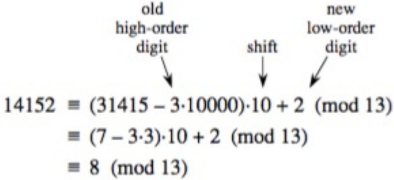
- What happens if p and t_s become too large?
- Solution:
  - Compute p and all t_ss modulo a suitable modulus q.



  - For a d-ary alphabet $\{0, 1, \cdots, d-1\}$, we choose q so that dq fits within a computer word and adjust the recurrence equation to work modulo q (where $h \equiv d^{m-1} (mod\ q)$):

$$t_{s+1} = (d(t_s - hT[s+1]) + T[s+m+1])\ mod\ q.$$





$(a + b) \bmod Q = ((a \bmod Q) + (b \bmod Q)) \bmod Q$

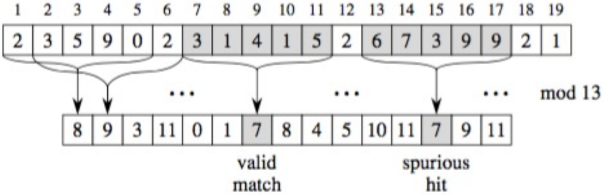
$(a * b) \bmod Q = ((a \bmod Q) * (b \bmod Q)) \bmod Q$

Rabin-Karp Algorithm

- Since the computation of p, t_0, and all values $t_1, t_2, \cdots, t_{n-m}$ can be performed modulo q, we can compute p modulo q in $\Theta(m)$ time and all the t_s's modulo q in $\Theta(n-m+1)$ time.
- Another Problem: working modulo q is not perfect, since $t_s \equiv p \ (mod\ q)$ does not imply that $t_s = p$.
- On the other hand, if $t_s \not\equiv p \ (mod\ q)$, then we definitely have that $t_s \neq p$, so that shift s is invalid.
- We can thus use the test $t_s \equiv p \ (mod\ q)$ as a fast heuristic test to rule out invalid shifts s.

Rabin-Karp Algorithm

- Any shift s for which $t_s \equiv p \ (mod\ q)$ must be tested further to see if s is really valid or we just have a spurious hit.



- This testing can be done by explicitly checking the condition $P[1 \cdots m] = T[s+1 \cdots s+m]$.
- If q is large enough, then we can hope that spurious hits occur infrequently enough that the cost of the extra checking is low.

Rabin-Karp Algorithm

```
RABIN-KARP-MATCHER(T, P, d, q)
n ← length[T]
m ← length[P]
h ← d^(m−1) mod q
p ← 0
t_0 ← 0
for i ← 1 to m                          ▷ Preprocessing.
      do p ← (dp + P[i]) mod q
         t_0 ← (dt_0 + T[i]) mod q
for s ← 0 to n − m                      ▷ Matching.
      do if p = t_s
            then if P[1 . . m] = T[s + 1 . . s + m]
                     then print "Pattern occurs with shift" s
         if s < n − m
            then t_{s+1} ← (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
```

Finite State Automata (Review)

Definition (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.

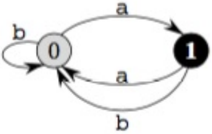
- The finite automaton begins in state q_0 and reads the characters of its input string one at a time.
- If the automaton is in state q and reads input character a, it moves (makes a transition) from state q to state $\delta(q, a)$.
- Whenever its current state q is a member of A, the machine M is said to have accepted the string read so far. An input that is not accepted is said to be rejected.

Finite State Automata (Review)

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

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

	input	
state	a	b
0	1	0
1	0	0



String-Matching Automata

Definition (suffix function)

A suffix function σ corresponding to pattern $P[1 \cdots m]$ is a mapping from Σ^* to $\{0, 1, \cdots, m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is a suffix of x:

$$\sigma(x) = \max\{k \mid P_k \sqsupset x\}.$$

Example

For the pattern $P = ab$, we have $\sigma(\epsilon) = 0$, $\sigma(ccaca) = 1$, and $\sigma(ccab) = 2$.

- For a pattern P of length m, we have $\sigma(x) = m$ iff $P \sqsupset x$.
- If $x \sqsupset y$, then $\sigma(x) \leqslant \sigma(y)$ (following from the definition of the suffix function).

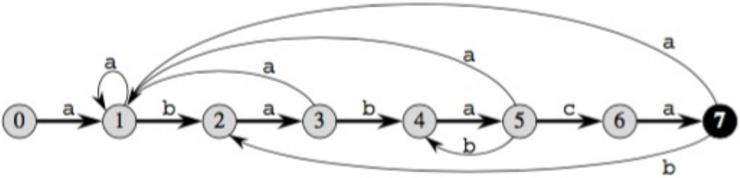
String-Matching Automata

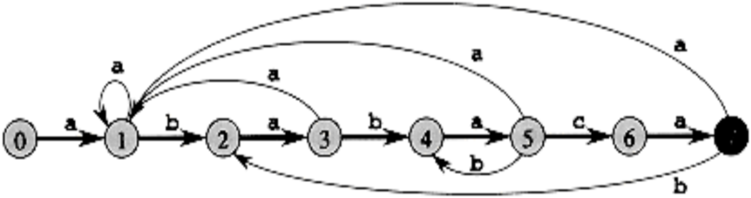
Constructing the String-Matching Automata

For a given pattern $P[1 \cdots m]$, the corresponding string-matching automaton would be as follows:

- $Q = \{0, 1, \cdots, m\}$.
- $q_0 = 0$.
- $A = \{m\}$.
- The transition function δ is defined by the following equation, for any state q and character a:

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





(a)

state	input a	input b	input 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)

String-Matching Automata

- The machine maintains as an invariant of its operation that $\varphi(T_i) = \sigma(T_i)$ (will be proved later).
- This means that after scanning T_i, the machine is in state $\varphi(T_i) = q$, where $q = \sigma(T_i)$ is the length of the longest suffix of T_i that is also a prefix of the pattern P.
- If the next character scanned is $T[i+1] = a$, then the machine should make a transition to state $\sigma(T_{i+1}) = \sigma(T_i a)$.
- The later proof shows that $\sigma(T_i a) = \sigma(P_q a)$, i.e. to compute the length of the longest suffix of $T_i a$ that is a prefix of P, we can compute the longest suffix of $P_q a$ that is a prefix of P.
- Therefore, setting $\delta(q, a) = \sigma(P_q a)$ maintains the desired invariant.

String-Matching Automata (matcher)

If the string-matching automaton is constructed (as a preprocess) for the pattern P, then the following algorithm could be used as a matcher.

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

- Time Complexity: $\Theta(n)$.

String-Matching Automata (transition function)

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

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

- Time Complexity: $O(m^3|\Sigma|)$.
- This Complexity can be reduced to $O(m|\Sigma|)$. How?

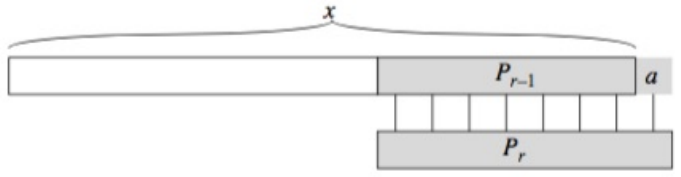
String-Matching Automata (Correctness)

Lemma (Suffix-function inequality)

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

Proof.

Let $r = \sigma(xa)$ and follow the figure...



$\square$

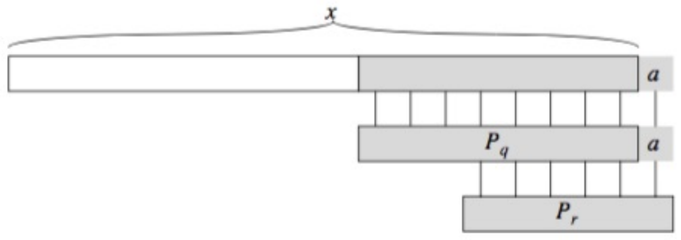
String-Matching Automata (Correctness)

Lemma (Suffix-function recursion)

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

Proof.

Let $r = \sigma(xa)$ and follow the figure...



$\square$

String-Matching Automata (Correctness)

Theorem

If φ is the final-state function of a string-matching automaton for a given pattern P and $T[1 \cdots n]$ is an input text for the automaton, then $\varphi(T_i) = \sigma(T_i)$ for $i = 0, 1, \cdots, n$.

Proof.

The proof is by induction on i. For $i = 0$, the theorem is trivially true, since $T_0 = \epsilon$. Thus, $\varphi(T_0) = 0 = \sigma(T_0)$. Now, we assume that $\varphi(T_i) = \sigma(T_i)$ and prove that $\varphi(T_{i+1}) = \sigma(T_{i+1})$. Let q denotes $\varphi(T_i)$, and let a denotes $T[i+1]$. Then:

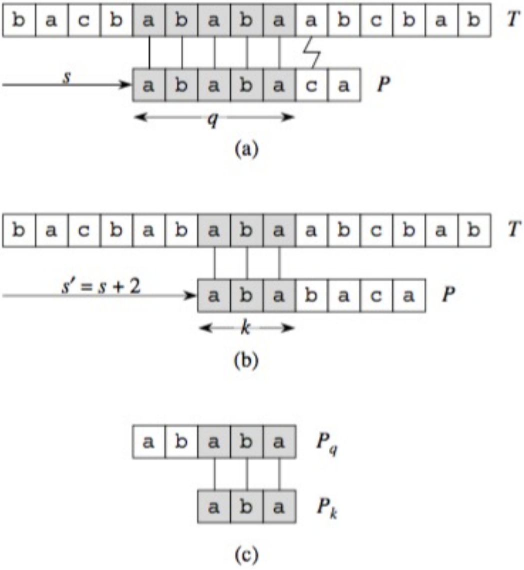
$$
\begin{aligned}
\varphi(T_{i+1}) &= \varphi(T_i a) && \text{(by the definitions of } T_{i+1} \text{ and } a\text{)} \\
&= \delta(\varphi(T_i), a) && \text{(by the definition of } \varphi\text{)} \\
&= \delta(q, a) && \text{(by the definition of } q\text{)} \\
&= \sigma(P_q a) && \text{(by the definition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

$\square$

The Knuth-Morris-Pratt (KMP) algorithm

- This algorithm avoids the computation of the costly transition function δ.
- Instead, it uses an auxiliary function $\pi[1 \cdots m]$ (called Prefix Function), precomputed from the pattern in time $\Theta(m)$.
- For any state $q = 0, 1, \cdots, m$ and any character $a \in \Sigma$, the value $\pi[q]$ contains the information that is independent of a and is needed to compute $\delta(q, a)$.
- The prefix function π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself.
- The array π has only m entries, whereas δ has $m \times |\Sigma|$ entries.
- Its matching time would be $\Theta(n)$.

KMP Algorithm: Motivation



KMP Algorithm: Motivation

General Question

Given that pattern characters $P[1 \cdots q]$ match text characters $T[s+1 \cdots s+q]$, what is the least shift $s' > s$ such that

$$P[1 \cdots k] = T[s'+1 \cdots s'+k],$$

where $s' + k = s + q$?

- Such a shift s' is the first shift greater than s that is not necessarily invalid due to our knowledge of $T[s+1 \cdots s+q]$.
- In the best case, we have that $s' = s + q$, and shifts $s+1, s+2, \cdots, s+q-1$ are all immediately ruled out.
- In any case, at the new shift s' we don't need to compare the first k characters of P with the corresponding characters of T, since we are guaranteed that they match.

KMP Algorithm: prefix function

- The necessary information can be precomputed by comparing the pattern against itself.
- Since $T[s' + 1 \cdots s' + k]$ is part of the known portion of the text, it is a suffix of the string P_q.
- Equation $P[1 \cdots k] = T[s' + 1 \cdots s' + k]$ can therefore be interpreted as asking for the largest $k < q$ such that $P_k \sqsupset P_q$.
- Then, $s' = s + (q - k)$ is the next potentially valid shift.

Formal definition of prefix function

Given a pattern $P[1 \cdots m]$, the prefix function for the pattern P is the function $\pi : \{1, 2, \cdots, m\} \mapsto \{0, 1, \cdots, m-1\}$ such that

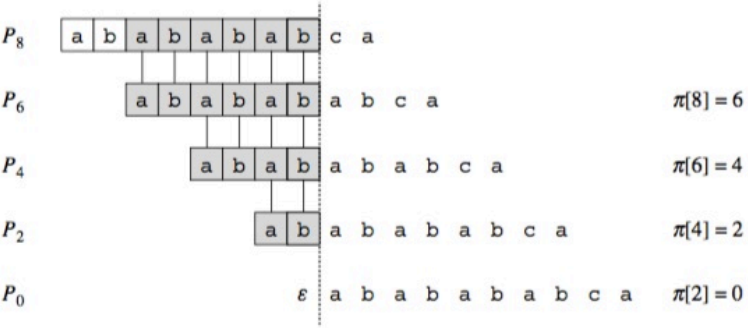
$$\pi[q] = \max\{k \mid k < q \text{ and } P_k \sqsupset P_q\}.$$

One again: $\pi[q]$ is the length of the longest prefix of P that is a suffix of P_q.

KMP Algorithm: prefix function

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

(a)



(b)

KMP Algorithm: Matcher

```
KMP-MATCHER(T, P)
n ← length[T]
m ← length[P]
π ← COMPUTE-PREFIX-FUNCTION(P)
q ← 0                                  ▷ Number of characters matched.
for i ← 1 to n                         ▷ Scan the text from 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 the next match.
```

- Time Complexity: $\Theta(n)$ (Amortized analysis?)

KMP Algorithm: Computing Prefix Function

```
COMPUTE-PREFIX-FUNCTION(P)
m ← length[P]
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
```

- Time Complexity: $\Theta(m)$ (Amortized analysis?)

Exercises

1. Show how to extend the Rabin-Karp method to handle the problem of looking for a given $m \times m$ pattern in an $n \times n$ array of characters. (The pattern may be shifted vertically and horizontally, but it may not be rotated)
2. Draw a state-transition diagram for a string-matching automaton for the pattern *ababbabbababbababbabb* over the alphabet $\{a, b\}$.
3. Given two patterns P and P', describe how to construct a finite automaton that determines all occurrences of either pattern. Try to minimize the number of states in your automaton.
4. Compute the prefix function π for the pattern *ababbabbabbababbabb* when the alphabet is $\Sigma = \{a, b\}$.
5. Give a linear-time algorithm to determine if a text T is a cyclic rotation of another string T' . For example, *arc* and *car* are cyclic rotations of each other.
6. Give an efficient algorithm for computing the transition function δ for the string-matching automaton corresponding to a given pattern P. Your algorithm should run in time $O(m|\Sigma|)$. (Hint: Prove that $\delta(q, a) = \delta(\pi[q], a)$ if $q = m$ or $P[q + 1] \neq a$.)

End.