



Theory of algorithms (10th lecture)

Pál Pusztai
pusztai@sze.hu

Outline

- The string-matching problem
 - Notation and terminology
 - A naive string matcher
 - The Rabin-Karp matcher
 - String matching with finite automata
 - Computing the transition function
 - Matching
 - The Knuth-Morris-Pratt matcher
 - Computing the prefix function
 - Matching
- Exercises



String matching

■ The string-matching problem

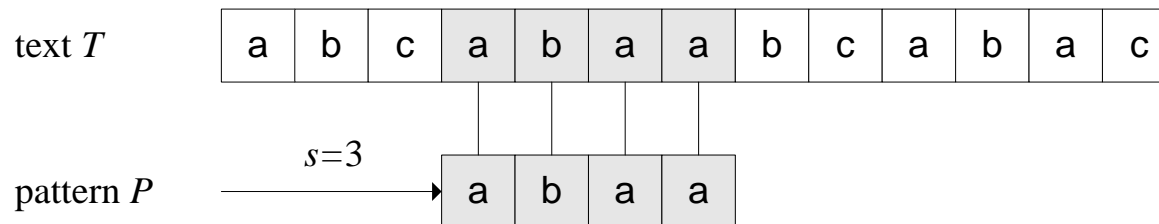
Let us assume that the **text** is an array $T[1..n]$ of length n and that the **pattern** is an array $P[1..m]$ of length $m \leq n$. The elements of P and T are characters drawn from a **finite alphabet** Σ .

Possible alphabets: $\Sigma = \{0, 1\}$, $\Sigma = \{a, b, \dots, z\}$. The character arrays P and T are often called **strings** of characters.

Pattern P **occurs with shift** s in text T (or, equivalently, the pattern P **occurs beginning at position** $s+1$ in text T) if $0 \leq s \leq n-m$ and $T[s+1..s+m] = P[1..m]$ (that is, if $T[s+j] = P[j]$, for $1 \leq j \leq m$).

If P occurs with shift s in T , then s is a **valid shift**, otherwise, s is an **invalid shift**.

The **string-matching problem**: to find all valid shifts of a given pattern P in a given text T .



An example of the string-matching problem

String matching

■ Notation and terminology

The Σ^* denotes the set of all finite-length strings formed using characters from the alphabet Σ .

The zero-length **empty string**, denoted ε , also belongs to Σ^* .

The length of a string x is denoted $|x|$.

The **concatenation** of two strings x and y , denoted xy , has length $|x| + |y|$ and consists of the characters from x followed by the characters from y .

A string w is a **prefix** of a string x , denoted $w \sqsubset x$, if $x = wy$ for some string $y \in \Sigma^*$.

A string w is a **suffix** of a string x , denoted $w \sqsupset x$, if $x = yw$ for some string $y \in \Sigma^*$.

Notes: If $w \sqsubset x$ or $w \sqsupset x$ then $|w| \leq |x|$. The empty string ε is both a suffix and a prefix of every string.

Example: $ab \sqsubset abcca$, $cca \sqsupset abcca$.

Let P_k denote the k -character prefix $P[1..k]$ of the pattern $P[1..m]$. Thus, $P_0 = \varepsilon$ and $P_m = P[1..m] = P$.

Let T_k denote the k -character prefix $T[1..k]$ of the text $T[1..n]$.

The **string-matching problem** (using this notation): to find all shifts s in the range $0 \leq s \leq n - m$ such that $P \sqsupset T_{s+m}$.



String matching

NAIVE-STRING-MATCHER(T, P)

```
1   $n = T.length$ 
2   $m = P.length$ 
3  for  $s = 0$  to  $n - m$ 
4      if  $P[1..m] == T[s+1..s+m]$ 
5          write „Pattern occurs with shift ”,  $s$ 
```

Efficiency: It takes time $O((n-m+1)m)$. The running time equals the matching time because it requires no preprocessing.

This **brute-force** algorithm is inefficient because it entirely ignores information gained about the text for one value of s when it considers other values of s . For example, if $P = aaab$ and we find that $s = 0$ is valid, then none of the shifts 1, 2, or 3 are valid, since $P[4] = b$.

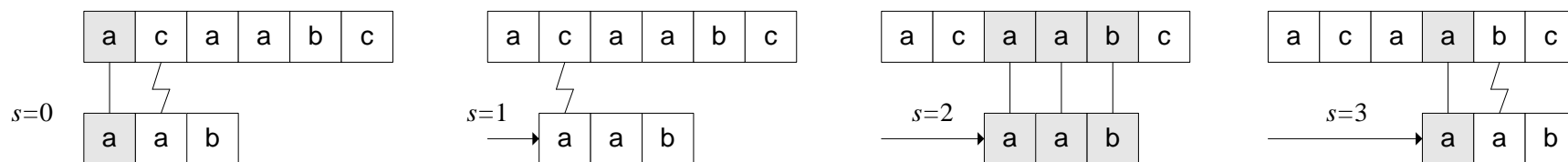
In our pseudocode, we allow two equal-length strings to be compared for equality as a primitive operation.

If the strings are compared from left to right and the comparison stops when a mismatch is discovered, we assume that the time taken by such a test is a linear function of the number of matching characters discovered.

To be precise, the test $x == y$ is assumed to take time $\Theta(t+1)$, where t is the length of the longest string z such that $z \sqsubset x$ and $z \sqsubset y$.



String matching



The operation of NAIVE-STRING-MATCHER

Exercises

- What valid shift values s are the results of NAIVE-STRING-MATCHER for the pattern $P=0001$ in the text $T=000010001010001$?
- Suppose that all characters in the pattern P are different. Show how to accelerate NAIVE-STRING-MATCHER to run in time $O(n)$ on an n -character text T .



String matching

Let us assume that $\Sigma = \{0, 1, 2, \dots, 9\}$, so that each character is a decimal digit. We can then view a string of k consecutive characters as representing a length- k decimal number.

Given a pattern $P[1..m]$, let p denote its corresponding decimal value. In a similar manner, given a text $T[1..n]$, let t_s denote the decimal value of the length- m substring $T[s+1..s+m]$, for $s=0, 1, 2, \dots, n-m$. Certainly, $t_s=p$ if and only if $T[s+1..s+m] = P[1..m]$, thus, s is a valid shift if and only if $t_s=p$.

We can compute p in time $\Theta(m)$ using **Horner's rule**:

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \dots + 10(P[2] + 10(P[1])) \dots).$$

Similarly, t_0 can be computed from $T[1..m]$ in time $\Theta(m)$.

The remaining values t_1, t_2, \dots, t_{n-m} can be computed in time $\Theta(n-m)$, as t_{s+1} can be computed from t_s in constant time, since:

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1].$$

Therefore, all occurrences of the pattern $P[1..m]$ in the text $T[1..n]$ can be found with $\Theta(m)$ preprocessing time and $\Theta(n-m+1)$ matching time.

Example: If $m=5$ and $t_s=31415$, then we wish to remove the high-order digit $T[s+1]=3$ and bring in the new low-order digit (suppose it is $T[s+5+1]=2$) to obtain

$$t_{s+1} = 10(31415 - 10000 \cdot 3) + 2 = 14152.$$



String matching

Problem: p and t_s may be too large to work with conveniently.

Solution: computing p and the t_s values modulo a suitable modulus q .

In general, with a d -ary alphabet $\{0, 1, \dots, d-1\}$, we choose q so that dq fits within a computer word.

Computing:

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1] \quad (\text{Decimal alphabet})$$

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q, \quad (d\text{-ary alphabet and modulo } q)$$

where $h = d^{m-1} \bmod q$ is the value of the digit „1” in the high-order position of an m -digit text window.

Problem: $t_s \equiv p \pmod{q}$ does not imply that $t_s = p$. On the other hand, if the equation is not satisfied, then we definitely have that $t_s \neq p$, so that shift s is invalid.

Solution: Any shift s for which $t_s \equiv p \pmod{q}$ must be tested further to see whether s is really valid or we just have a **spurious hit**. This additional test explicitly checks the condition $P[1..m] = T[s+1..s+m]$.

If q is large enough, then spurious hits expectedly occur infrequently enough that the cost of the extra checking is low.

String matching

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

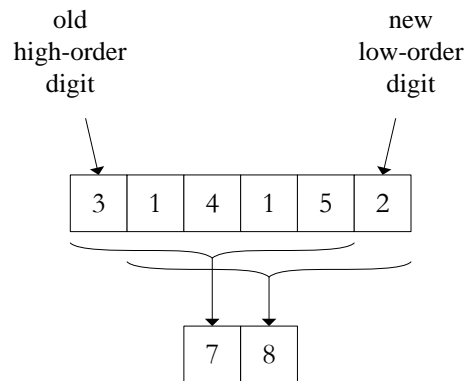
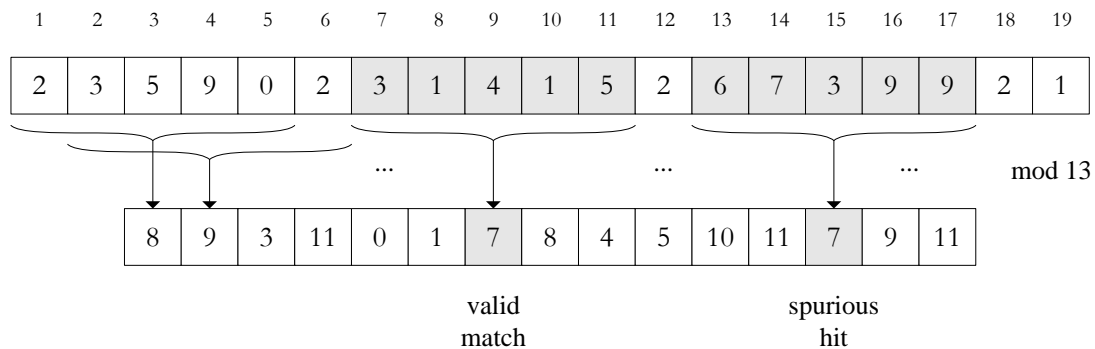
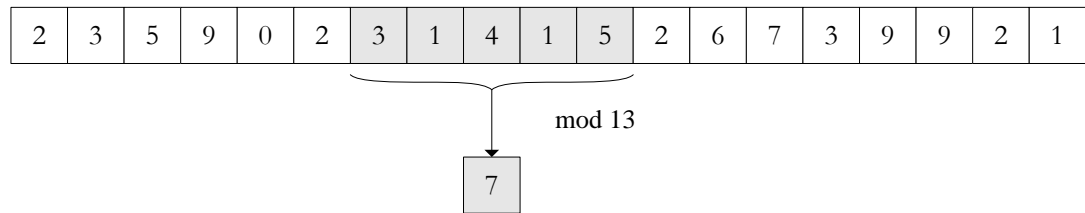
```
1   $n = T.length$ 
2   $m = P.length$ 
3   $h = d^{m-1} \bmod q$ 
4   $p = 0$ 
5   $t_0 = 0$ 
6  for  $i = 1$  to  $m$                                 // preprocessing
7       $p = (dp + P[i]) \bmod q$ 
8       $t_0 = (dt_0 + T[i]) \bmod q$ 
9  for  $s = 0$  to  $n - m$                                 // matching
10     if  $p == t_s$ 
11         if  $P[1..m] == T[s+1..s+m]$ 
12             write „Pattern occurs with shift ”,  $s$ 
13     if  $s < n - m$ 
14          $t_{s+1} = (d(t_s - T[s+1])h + T[s+m+1]) \bmod q$ 
```

Efficiency: It takes $\Theta(m)$ preprocessing time, and its matching time is $O((n-m+1)m)$.

Remarks: If the expected number of valid shifts is small ($O(1)$) and we choose the prime q to be larger than the length of the pattern, then we can expect $O(n)$ matching time.



String matching



old high-order digit shift new low-order digit

$$\begin{aligned}
 14152 &\equiv (31415 - 3 \cdot 10000) \cdot 10 + 2 \pmod{13} \\
 &\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13} \\
 &\equiv 8 \pmod{13}
 \end{aligned}$$

The operation of RABIN-KARP-MATCHER

Exercises

- Can the indices of t be omitted in RABIN-KARP-MATCHER?
- What is the worst case of RABIN-KARP-MATCHER?
- Working modulo $q=11$, how many spurious hits does the RABIN-KARP-MATCHER encounter in the text $T=314159265$ when looking for the pattern $P=26$?



String matching

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 **operation** of a finite automaton:

- The finite automaton begins in state q_0 .
- It 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 has **accepted** the string read so far. An input that is not accepted is **rejected**.



String matching

A finite automaton M induces a function \varnothing , called the **final-state function**:

$\varnothing : \Sigma^* \rightarrow Q$, $\varnothing(w)$ is the state M ends up in after scanning the string w .

M accepts a string w if and only if $\varnothing(w) \in A$.

The recursive definition of \varnothing with the transition function δ :

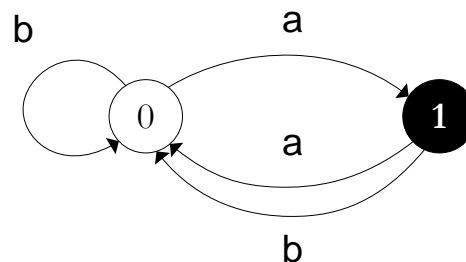
$$\varnothing(\varepsilon) = q_0,$$

$$\varnothing(wa) = \delta(\varnothing(w), a) \quad (w \in \Sigma^*, a \in \Sigma).$$

The transition function δ

state	input	
	a	b
0	1	0
1	0	0

The state-transition diagram



A simple two-state finite automaton

String matching

For a given pattern P , a string-matching automaton is constructed in a preprocessing step before using it to search the text string:

- An auxiliary function σ , called the **suffix function** is defined corresponding to pattern $P[1..m]$.
 - $\sigma : \Sigma^* \rightarrow \{0, 1, \dots, m\}$, such that $\sigma(x)$ is the length of the longest prefix of P that is also a suffix of x : $\sigma(x) = \max\{k: P_k \sqsupset x\}$.
 - Example: If $P=ab$, then $\sigma(\varepsilon)=0$, $\sigma(ccaca)=1$, $\sigma(ccab)=2$.
 - **Properties** of σ :
 - For a pattern P of length m , we have $\sigma(x)=m$ if and only if $P \sqsupset x$.
 - If $x \sqsupset y$, then $\sigma(x) \leq \sigma(y)$.
- 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 :
$$\delta(q, a) = \sigma(P_q a)$$

Remark: It comes from that the automaton maintains the following invariant: $\sigma(T_i) = \sigma(T_i)$.



String matching

COMPUTE-TRANSITION-FUNCTION(P, Σ)

```
1   $m = P.length$ 
2  for  $q = 0$  to  $m$ 
3      for each character  $a \in \Sigma$ 
4           $k = \min(m+1, q+2)$ 
5          repeat
6               $k = k-1$ 
7          until  $P_k \supseteq P_q a$ 
8           $\delta(q, a) = k$ 
9  return  $\delta$ 
```

Efficiency: The running time is $O(m^3|\Sigma|)$, because the outer loops contribute a factor of $m|\Sigma|$, the inner **repeat** loop can run at most $m+1$ times, and the test $P_k \supseteq P_q a$ on line 7 can require comparing up to m characters.

Remark: δ can be computed in $O(m|\Sigma|)$ time (by utilizing some cleverly computed information about the pattern P).



Exercises

- Give the transition function δ and the state-transition diagram of the string-matching automaton for the pattern $P = \text{aabab}$ over the alphabet $\Sigma = \{a, b\}$.



String matching

FINITE-AUTOMATON-MATCHER(T, δ, m)

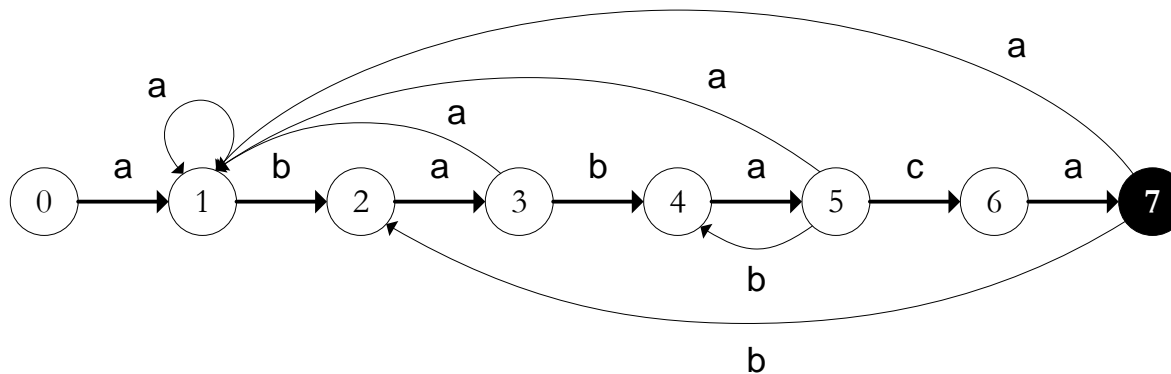
```
1   $n = T.length$ 
2   $q = 0$ 
3  for  $i = 1$  to  $n$ 
4       $q = \delta(q, T[i])$ 
5      if  $q == m$ 
6          write „Pattern occurs with shift ”,  $i-m$ 
```

Efficiency: The running time on a text string of length n is $\Theta(n)$.

Summarizing: With the improved procedure for computing δ , all occurrences of a length- m pattern in a length- n text over an alphabet Σ can be found with $O(m|\Sigma|)$ preprocessing time and $\Theta(n)$ matching time.



String matching



state	input			P
	a	b	c	
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	

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

A string-matching automaton



String matching

- How can we accelerate the naive string matching?
 - Instead of shifting the pattern in the text by one, if possible „step over” the invalid shifts.
 - When we check a shift, we do not examine again the matched characters that we have already examined before.
- In general, it is useful to know the answer to the following question:
 - Given that pattern characters $P[1..q]$ match text characters $T[s+1..s+q]$, what is the least shift $s' > s$ such that for some $k < q$,
$$P[1..k] = T[s'+1..s'+k], \quad \text{where } s' + k = s + q?$$
- In other words, knowing that $P_q \supset T_{s+q}$, we want the longest proper prefix P_k of P_q that is also a suffix of T_{s+q} .
 - $s' = s + q - k$ is the first shift after s that is not necessarily invalid.
 - In the best case, $k = 0$, so that $s' = s + q$, and we immediately rule out shifts $s+1, s+2, \dots, s+q-1$.
 - 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 they match.
- We can precompute the prefix function π for a pattern P that encapsulates knowledge about how the pattern matches against shifts of itself.
 - With function π we can avoid testing useless shifts in the naive pattern-matching algorithm and precomputing the full transition function δ like for a string-matching automaton.



String matching

The **prefix function** for a pattern $P[1..m]$ is the function $\pi : \{1, 2, \dots, m\} \rightarrow \{0, 1, \dots, m-1\}$ such that $\pi[q] = \max\{k: k < q \text{ and } P_k \sqsupset P_q\}$.

That is, $\pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q .

COMPUTE-PREFIX-FUNCTION(P)

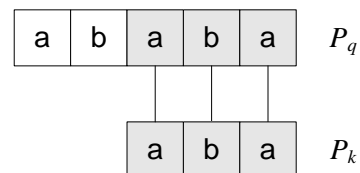
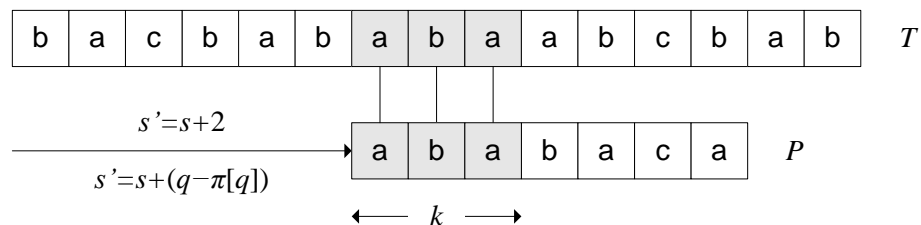
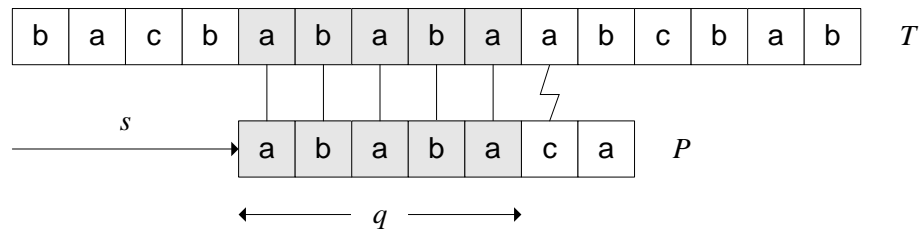
```
1   $m = P.length$ 
2  let  $\pi[1..m]$  be a new array
3   $\pi[1] = 0$ 
4   $k = 0$ 
5  for  $q = 2$  to  $m$ 
6      while  $k > 0$  and  $P[k+1] \neq P[q]$ 
7           $k = \pi[k]$ 
8      if  $P[k+1] == P[q]$ 
9           $k = k+1$ 
10      $\pi[q] = k$ 
11 return  $\pi$ 
```

Efficiency: It takes time $\Theta(m)$, as the inner loop requires $O(1)$ time (see amortized analysis in the textbook).



String matching

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



The prefix function π and its using

Exercises

- Give the prefix function π for the pattern $P = \text{ababbabbababbabbabb}$ over alphabet $\Sigma = \{a, b\}$.



String matching

KMP-MATCHER(T, P)

```
1   $n = T.length$ 
2   $m = P.length$ 
3   $\pi = \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4   $q = 0$  // number of characters matched
5  for  $i = 1$  to  $n$  // scan the text from left to right
6      while  $q > 0$  and  $P[q+1] \neq T[i]$ 
7           $q = \pi[q]$  // next character does not match
8      if  $P[q+1] == T[i]$ 
9           $q = q+1$  // next character matches
10     if  $q == m$  // is all of  $P$  matched?
11         write „Pattern occurs with shift ”,  $i-m$ 
12          $q = \pi[q]$  // look for the next match
```

Efficiency: Preprocessing (computing π) takes time $\Theta(m)$, the matching time is $\Theta(n)$ (as the inner loop requires $O(1)$ time, similarly as in the previous algorithm).

Remark: The abbreviation KMP is given by the initials of the names Knuth-Morris-Pratt.



Exercises

- What values are assigned to variable q in running of KMP-MATCHER, if $T=bbababababba$, $P=ababa$ and $\Sigma=\{a, b\}$?



String matching

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

Efficiency of the string-matching algorithms

