ICS 311 #23: String Matching

Spring 2014:

This will be presented in half-day "special topic" format. There will be no screencast, quiz or problem set. We will have a brief lecture followed by an in-class activity, based on sections 32.1 (the naive algorithm) and 32.3 (string matching with finite state automata), both of which are reviewd in these notes.

While you are not likely to need to design or implement a string matching algorithm (they are built into modern software tools), questions about string matching algorithms reportedly still show up in job interviews (study KMP section 32.4 for this). It is also worth being familiar with this topic for the concept of finite-state automata as a form of table-driven algorithm, and how algorithms can be written to write other algorithms (the FSA).

Outline

Readings and Screencasts

- CLRS 3rd Ed. Chapter 32, with emphasis on KMP
- Screencasts not available

String Matching Introduction

The **string matching problem** is the problem of finding all occurrences of a string P (the **pattern**) in a target text T, also a string.

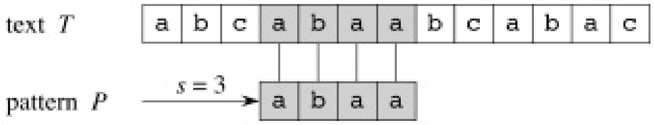
We treat both T and P as arrays $T[1 .. n]$ and $P[1 .. m]$, where $m \leq n$. The elements of the arrays are members of a finite alphabet Σ.

More precisely, we say that P occurs with **shift** s in T (P occurs beginning at position $s + 1$ in T) if

$0 \leq s \leq n - m$ *(one cannot fall off the end of T)*, and
$T[s + 1 .. s + m] = P[1 .. m]$.

The **string matching problem** is the problem of finding ***all valid shifts***.

A simple example:



Applications

String matching is obviously important for implementing search in ***text editing*** programs, and for ***document search***, whether searching within a single document or searching multiple documents for a topic of interest. This latter application extends to ***internet search engines***. Other specialized applications include processing ***DNA sequences*** in bioinformatics.

Brief History

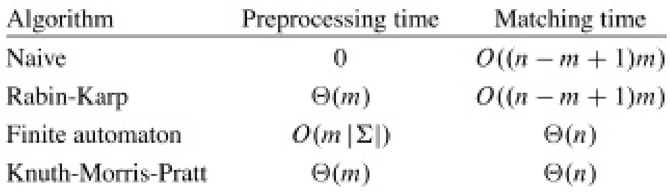
In 1970 Cook proved that an O(M+N) machine is possible. His theorem was not intended to be practical, but Knuth and Pratt followed the structure of the theorem to design an algorithm. Around the same time, Morris derived the algorithm independently as part of his design of a text editor. Eventually they found out about each other's work and published it in 1976. Also in 1976, Boyer and Moore found another efficient algorithm not covered here: see the Sedgewick book. In 1980 Rabin & Karp came up with a modification to brute force that searches on chunks of text of size M using a hash function. Researchers continue to come up with new algorithms.

Algorithms

There are a surprising number of variations on string searching algorithms. See http://www-igm.univ-mlv.fr/~lecroq/string/index.html for summary descriptions, code and animations in Java. Find out what a "**Backward Nondeterministic Dawg Matching**" algorithm is!

Most algorithms require **preprocessing** of the pattern before entering the **matching** phase. Analysis must consider both costs, and applications must determine whether the preprocessing is worth the speedup in matching.

The algorithms we cover are summarized by this table from the text:



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

Today we will cover how these algorithms work, but not any proofs of correctness, which you may find in the text.

Notation and Terminology

Prefix: $p \sqsubset t$ (p is a prefix of t) if $t = pw$ for some $w \in \Sigma^*$

Suffix: $s \sqsupset t$ (s is a suffix of t) if $t = ws$ for some $w \in \Sigma^*$

The **empty string** is denoted ε.

The ***k*-character prefix** $T[1 .. k]$ of any text or pattern T is denoted T_k.

Comment on String Matching Time: The test of whether "$x == y$" takes $\Theta(t + 1)$ time, where t is the length of the longest string z such that $z \sqsubset x$ and $z \sqsubset y$. (The "1" is included to cover the case where $t = 0$, since a positive amount of time must be expended to determine this fact.) This comparison loop will be implicit in some of our pseudocode.

Naive (Brute Force) String Matching

It is often instructive to start with a brute force algorithm, that we can then examine for possible improvements and also use as a baseline for comparison.

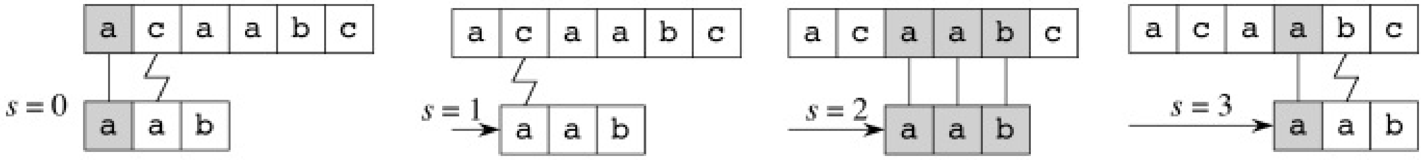
The obvious approach is to start at the first character of T, $T[1]$, and then step through T and P together, checking to see whether the characters match.

Once P has been match at any given shift $T[s]$, then go on to checking at $T[s+1]$ (since we are looking for all matches), up until $s = |T| - |P|$ (which is $n - m$).

```
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
```

Example

Suppose P = aab and T = acaabc. There are four passes:



You an see C code and a Java applet animation at http://www-igm.univ-mlv.fr/~lecroq/string/node3.html

Analysis

No preprocessing is required.

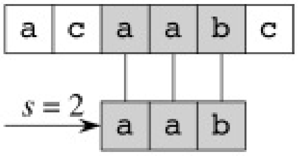
For each of $n - m + 1$ start positions, potentially m pattern characters are compared to the target text in the matching phase. Thus, the naive algorithm is $\mathbf{O((n - m + 1)m)}$ in the worst case.

```
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
```

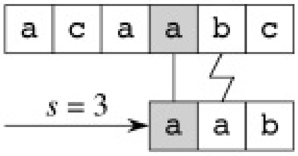
Inefficiencies

The brute force method does not use information about what has been matched, and does not use information about recurrences within the pattern itself. Consideration of these factors leads to improvements.

For example, when we matched P = aab at $s = 2$, we found that $T[5]$ = b:



Then it is not possible for a shift of $s = 3$ (or $s = 4$ if T were longer) to be valid, beause these shifts juxtapose $P[2]$ = a (and $P[1]$ = a if applicable) against $T[5]$ = b:



This information is used in the finite state automata and Knuth-Morris-Pratt approaches.

Matching with Finite State Automata

Finite state automata are machines that have a finite number of states, and move between states as they read input one symbol at a time.

Algorithms that construct (or simulate) finite state automata can be very efficient matchers, but they require some preprocessing to construct.

Finite State Automata

Finite state automata are widely used in computability theory as well as in practical algorithms. It's worth knowing about them even if you are not doing string matching.

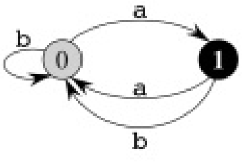
A **finite state automaton (FSA)** or **finite automaton** M is a 5-tuple (Q, q_0, A, Σ, δ) where

- ***Q*** is a finite set of ***states***
- $q_0 \in Q$ is the ***start state***
- $A \subseteq Q$ is a set of ***accepting states***
- **Σ** is a finite ***input alphabet***
- $\delta : Q \times \Sigma \rightarrow Q$ is the ***transition function*** of M.

The FSA starts in state q_0. As each character (symbol) of the input string is read, it uses δ to determine what state to transition into. Whenever M is in a state of A, the input read so far is **accepted**.

Here is a simple example: $q_0 = 0$ and $A = \{1\}$. (*What are Q, Σ, and δ?*)

	input	
state	a	b
0	1	0
1	0	0



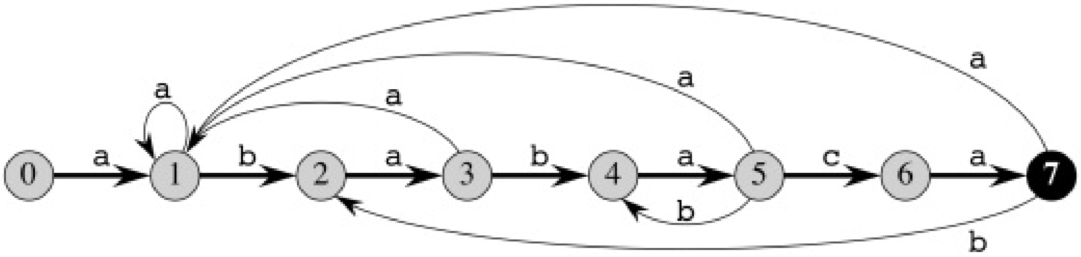
What strings does this FSA accept?

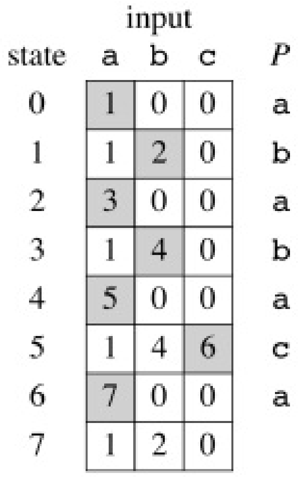
We define the **final state function** $\varphi : \Sigma^* \rightarrow Q$ such that $\varphi(w)$ is the final state M ends up in after reading w:

$$\varphi(\varepsilon) = q_0$$
$$\varphi(wa) = \delta(\varphi(w), a) \quad \text{for } w \in \Sigma^*, a \in \Sigma$$

String Matching Automata

Let's see how they work by an example. This is the string matching automaton for P = `ababaca`:





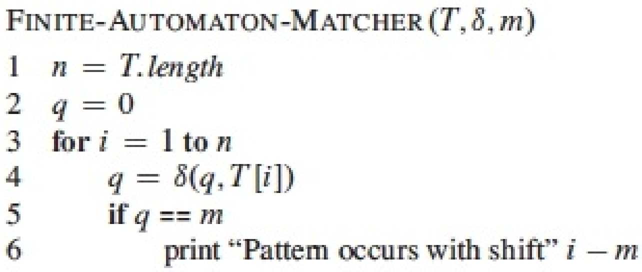
	input			
state	a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

The start state is 0 and the only accepting state is 7. Any transitions not shown (e.g., if `c` is read while in states 1, 2, 3, 4, 6, and 7) are assumed to go back to state 0.

This automaton can be represented with the table to the right. The shading shows the sequence of states through which a successful match transitions. These transitions correspond to the darkened arrows in the diagram.

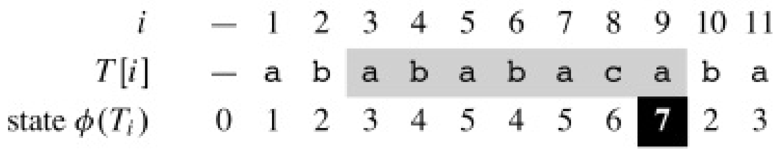
We can run this automaton continuously on a text T, and if and when state 7 is entered output the relevant positions: the match will start with shift $i - m$ or at position $i - m + 1$.

The following code simulates any FSA on input text T, given the FSA's table δ and pattern length m:



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

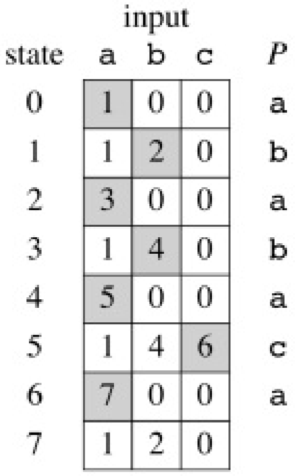
For example, below is a run on T = `abababacaba`, which includes P = `ababaca` starting at position 3: `ab ababaca ba`.



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

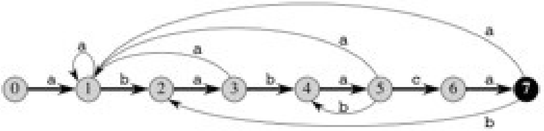
State 7 is reached at i = 9, so the pattern occurs starting at $i - m + 1$, or 9 − 7 + 1 = 3. The FSA keeps going after a match, as it may find other occurrences of the pattern.

Unlike the brute force approach, past work is not thrown away: the transitions following either failure to match a character or success to match the entire pattern pick up with the input read so far.



	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	

For example,



- *After Failure:* At $i = 5$, `ababa` has been matched in $T[1 .. 5]$ and `c` was expected but not found at $T[6]$. Rather than starting over, the FSA transitions to state $\delta(5, \texttt{b}) = 4$ to indicate that the pattern prefix P_4 = `abab` has matched the present text suffix $T[3 .. 6]$.

- *After Success:* At, $i = 9$, we are in state 7 (success), and `ab` is seen. We need not start over: the FSA transitions to state $\delta(7, \texttt{b}) = 2$ to reflect the fact that there is already a match to the prefix P_2 = `ab` at $T[9 .. 11]$.

This makes FSAs much more efficient than brute force in the matching phase. In fact, matching is $\Theta(n)$. But how do we build them?

Constructing Finite State Automata (Preprocessing Phase)

In general, the FSA is constructed so that the state number tells us how much of a prefix of *P* has been matched.

- If the pattern *P* is of length *m* and the FSA is in state *m*, then the pattern has been matched.

- If the state number is smaller than *m*, then the state number is the length of the prefix of *P* matched.

Another definition is needed to formalize this.

Definitions and Strategy

The **suffix function** corresponding to *P* of length *m* is $\sigma_P : \Sigma^* \to \{0, 1, ... m\}$ such that $\sigma_P(w)$ is the length of the longest prefix of *P* that is also a suffix of *x*:

$$\sigma_P(w) = \max\ \{k : P_k \sqsupset w\} .$$

For example, if *P* = `ab` then

- $\sigma(\varepsilon) = 0$
- $\sigma(\texttt{ccaca}) = 1$
- $\sigma(\texttt{ccab}) = 2$

(For simplicity, we leave out the subscript *P* when it is clear from the context.) Then we can define the automaton for pattern $P[1 .. m]$ as:

- $Q = \{0, 1 .. m\}$

- $q_0 = 0$

- $A = \{m\}$

- Σ is a superset of the characters in P
- $\boldsymbol{\delta(q, a) = \sigma(P_q\, a)}$ for any state q and character a. ($P_q\, a$ is the concatenation of the first q characters of P with the character a.)

By defining $\delta(q, a) = \sigma(P_q\, a)$, the state of the FSA keeps track of the longest prefix of the pattern P that has matched the input text T so far.

In order for a substring of T ending at $T[i]$ to match some prefix P_j, then this prefix P_j must be a suffix of $T[i]$.

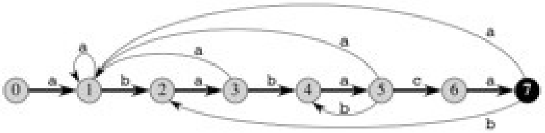
We design δ such that the state $q = \varphi(T[i])$ gives the length of the longest prefix of P that matches a suffix of T. We have:

> $q = \varphi(T_i) = \sigma(T_i)$, and $P_q \sqsupset T_i$ (P_q is a suffix of T_i).

Given a match so far to P_q (which may be ε) and reading character a, there are two kinds of transitions:

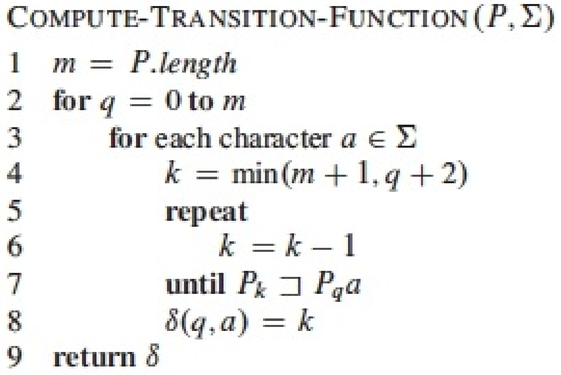
- When $a = P[q + 1]$, a continues to match the pattern, so $\delta(q, a) = q + 1$ (going along the dark "spine" arrows of the example).
- When $a \neq P[q + 1]$, a fails to match the pattern. The preprocessing algorithm given below ***matches the pattern against itself*** to identify the longest smaller prefix of P that is still matched.

An example of this second case was already noted above for $\delta(5, b) = 4$.



Preprocessing Procedure

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



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

The nested loops process all combinations of states q and characters a needed for the cells of the table representing δ.

Lines 4-8 set $\delta(q, a)$ to the largest k such that $P_k \sqsupset P_q a$.

- The preprocessor is *matching P against itself*.
- Thus, knowledge of the structure of P is used to retain information about the match so far, even when matches fail.
- By starting at the largest possible value of k (line 4) and working down (lines 5-7) we guarantee that we get the longest prefix of P that has been matched so far.
    - If the match succeeds at $k = q + 1$ then this transition indicates a successful match for the

current q and a.

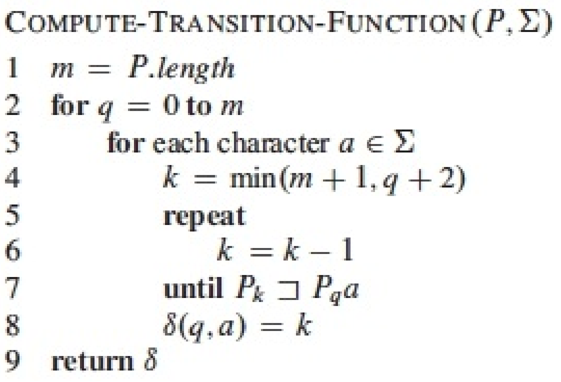
- The loop is guaranteed to end at $k = 0$, because $P_0 = \varepsilon$ is a suffix of any string.

Analysis

`Compute-Transition-Function` requires m*|Σ| for the nested outer loops.

Within these loops, the inner `repeat` runs at most $m + 1$ times; and the test on line 7 can require comparing up to m characters. Together lines 5-7 contribute O(m^2).

Therefore, `Compute-Transition-Function` is **O(m^3 |Σ|)**. This is rather expensive preprocessing, but the Θ(n) matching is the best that can be expected.



```
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 $P_k \sqsupset P_q a$
8          $\delta(q, a) = k$
9  return $\delta$
```

You an see C code and a Java applet animation at http://www-igm.univ-mlv.fr/~lecroq/string/nod43.html

Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt algorithm improves on the FSA algorithm by avoiding computation of δ.

Instead it precomputes an auxiliary **prefix function π_P**, represented as an array $\pi_P[1 .. m]$, that can be computed from a pattern P of length m in Θ(m) time. (We'll leave off the P subscript from now on.)

$\pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q. This information enables fast amortized computation of δ(q, a) on the fly.

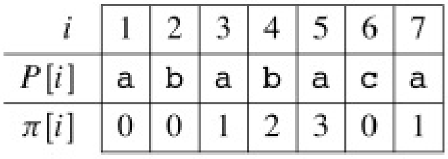
The KMP matcher operates like the FSA matcher, but using π instead of δ. There is some additional overhead at matching time, but asymptotic performance of matching remains Θ(n).

How π Works

Given $P[1 .. m]$, the prefix function π for P is π : {1, 2 ..., m} → {0, 1, ..., m-1} such that

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

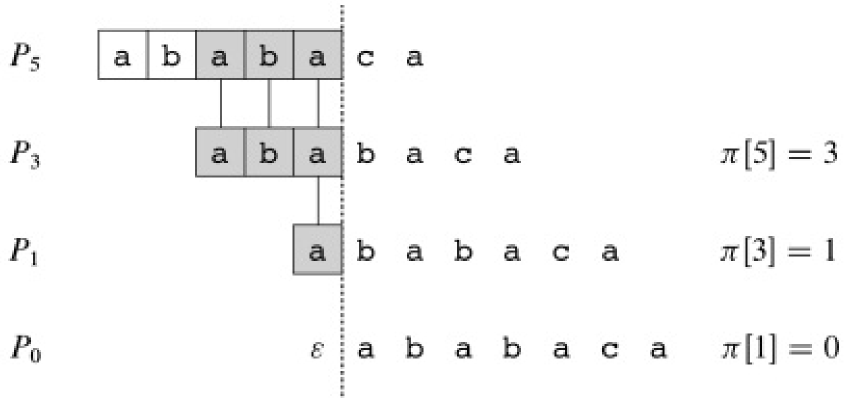
For example, for P = `ababaca`, π is as follows (*let's use the formula above to explain a few of the entries below, particularly for i=5.*):



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

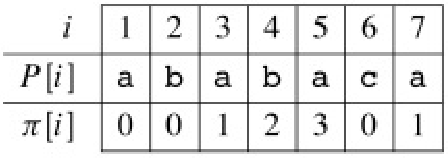
The table is structured such that a failure at a given position will drive the system to hop to the next smaller prefix that could be matched. (*Here I am skipping over the textbook's discussion of π^* and Lemma 32.5, and using examples instead*).

For example, for P = `ababaca`, if q = 5 and we are reading the 6th character (to the right of the line), consider what happens when that character is `c`, `b` or `a`:



From *P*[6] we see that a `c` is expected next.

- If a `c` is read, we go to state 6 and continue.
- If an `a` or `b` is read, then there is a mismatch and we need to figure out what prefix of *P* has still been matched. We look up π[5], which says to skip back to state 3.
- Repeating the test at state 3, the next character expected is *P*[4] = `b`. If we see a `b` we continue to state 4.
- Otherwise (the next character is `a`), we look up π[3] and skip back to state 1. Again, the next character does not match what is expected for state 1, and π[1] tells us to start over at state 0. In state 0, an `a` is expected, so it goes to state 1.



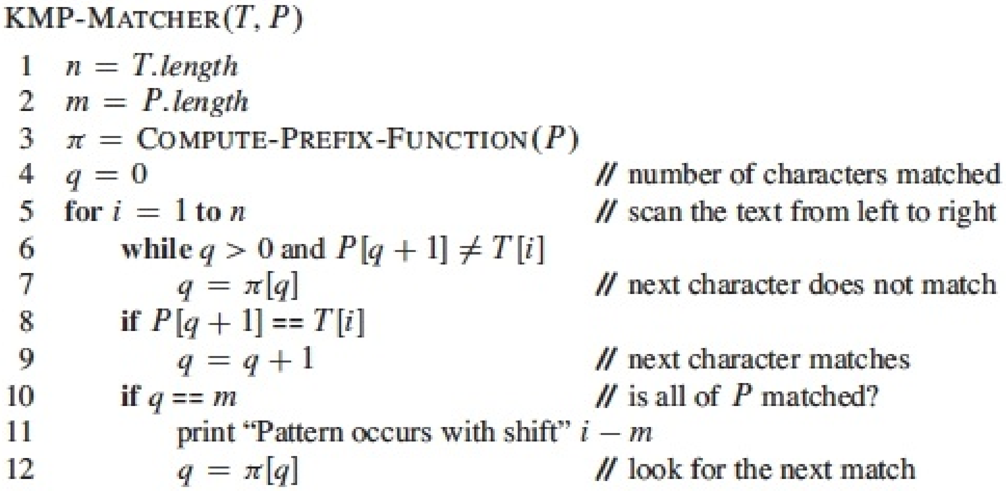
i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

Thus, π helps us retain information from prior comparisions, shifting back to the state for the maximum prefix matched so far, rather than starting over.

The KMP Matcher

Both the KMP matcher and the algorithm for computing π are similar to the FSA's `Compute-Transition-Function` (take the time to compare them when you read the text).

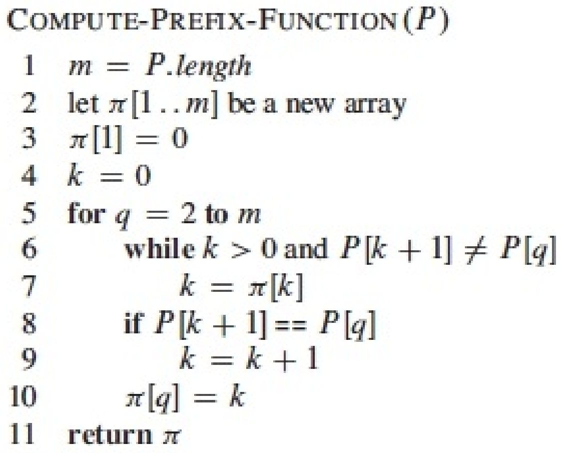
The KMP matching code is below. We can see the "skipping" discussed above happening in lines 6-7, with successful matching of the next character handled in lines 8-9. After a successful match, we jump to the appropriate state to continue looking for the next item in line 12.



```
KMP-MATCHER(T, P)
 1  $n = T.length$
 2  $m = P.length$
 3  $\pi$ = 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          print "Pattern occurs with shift" $i - m$
12          $q = \pi[q]$                 // look for the next match
```

Computing π

The code for computing π is very similar to that of `KMP-Matcher`, because the constructor is matching P against itself:



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

You an see C code and a Java applet animation at http://www-igm.univ-mlv.fr/~lecroq/string/node8.html

Analysis and Correctness

The text presents an analysis showing that `Compute-Prefix-Function` is $\mathbf{\Theta(m)}$ -- a considerable improvement over the FSA's $O(m^3 |\Sigma|)$ -- and `KMP-Matcher` is $\mathbf{\Theta(n)}$.

These are good results, but in practice the FSA approach is still used when a pattern will be used many times repeatedly, because of the greater simplicity of the `Finite-Automaton-Matcher` code.

The proof of correctness is accomplished by showing how KMP-Matcher simulates the FSA matcher's operation. It's worth reading the informal discussion pages 1009-1010, even if you don't wade into the formal proof.

Rabin-Karp Algorithm

There is an entirely different approach that functions in some respects like the brute force approach, but instead of testing character by character it tests on whole substrings of length m by using a hash function. If the hash function matches, it then uses brute force checking to verify the match.

Comparing Strings as Radix-*d* Numbers

Given $|\Sigma = d$, Rabin-Karp algorithm treats each string in Σ^* as if it were a number in radix d notation.

- For example, if $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ then $d = 10$, and we interpret the string "274" to have value $2 \cdot d^2 + 7 \cdot d^1 + 4 \cdot d^0 = 2 \cdot 100 + 7 \cdot 10 + 4 \cdot 1$.
- Similarly, if $\Sigma = \{a, b\}$ then $d = 2$, and we map the characters of $\{a, b\}$ to decimal values $\{0, 1\}$. Then "bab" = $1 \cdot d^2 + 0 \cdot d^1 + 1 \cdot d^0 = 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1$. (This is of course a binary representation of the decimal number 5.)
- Hexadecimal notation uses $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ and $d = 16$.

This idea can be extended to large but finite $|\Sigma|$.

Thus, we can treat $P[1 .. m]$ and substrings $T[s+1 .. s+m]$ $(0 \le s \le n - m)$ as m digit numbers in radix-$|\Sigma|$.

Our running example will assume $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and $d = 10$.

If we can compare these numbers directly to each other, we can quickly determine whether $P[1 .. m]$ matches $T[s+1 .. s+m]$.

Two modifications need to be made to make this method suitable for a string matching algorithm: shifting and hashing.

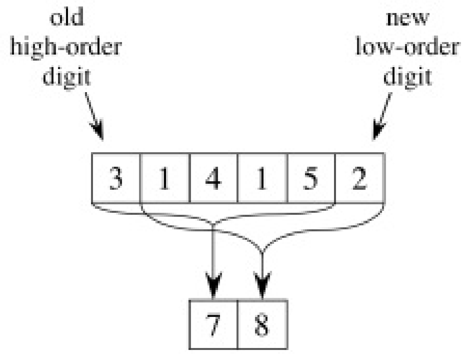
Shifting

If a match to $T[s+1 .. s+m]$ fails, we need to compute the next number for $T[s+\mathbf{2} .. s+m+1]$.

Rather than recomputing the entire sum, this can be done efficiently by a mathematical "shift":

- Subtract the value of the highest order digit $a \cdot d^{m-1}$.
- Multiply the remaining value by d to shift the values of the digits up by one position.
- Add the value of the new character $T[s + m + 1]$.

An example is shown below, shifting a 5-digit number. (The mod will be explained below.)



$$\begin{aligned} 14152 &\equiv (31415 - \overset{\text{old high-order digit}}{3 \cdot 10000})\overset{\text{shift}}{\cdot 10} + \overset{\text{new low-order digit}}{2} \pmod{13} \\ &\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13} \\ &\equiv 8 \pmod{13} \end{aligned}$$

Hashing

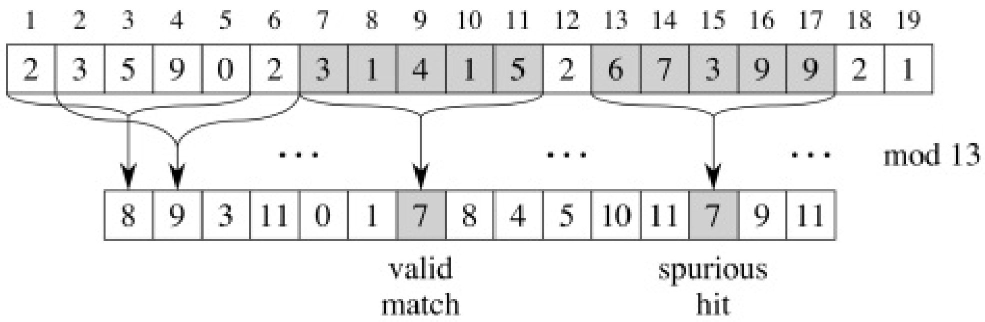
These numbers can get very large. Even on a machine that allows arbitrarily large numbers, this is still a problem: we can no longer assume that comparing two numbers takes constant time.

The solution is to compute the numbers modulo the largest prime number q such that dq still fits in one computer word (which can be manipulated with one machine level instruction).

Now you see why it is called "hashing:" it is like the division method of hashing.

For example, suppose we are matching to a 5 digit pattern $P = 31415$ in radix $d = 10$ we choose $q = 13$, and we are comparing to string 14142. The hash code for the pattern is 7, and for the target is 8, as shown in the previous example. The hashes do not match, so the two patterns cannot be the same.

However, the converse is not necessarily true. Hashing introduces a secondary problem: collisions. Two different numbers may hash to the same value, as shown in the example below.

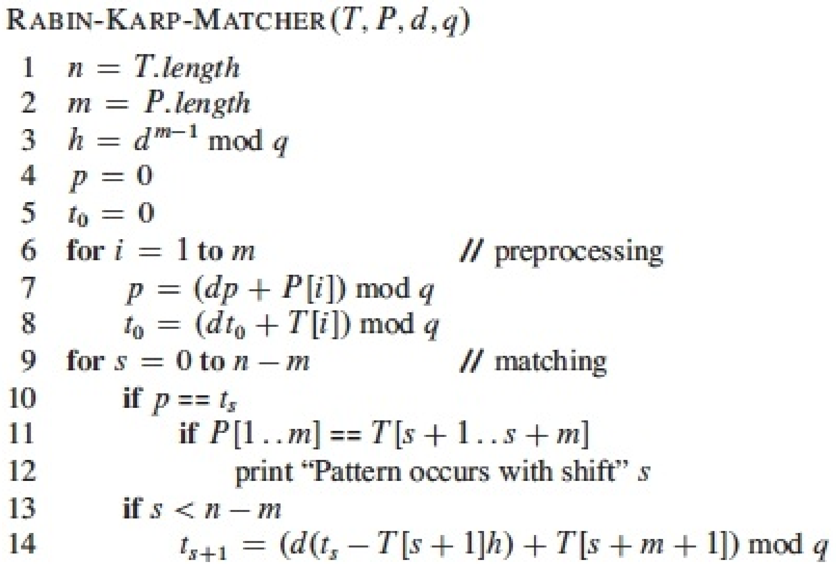


To solve this problem, when the hashes of $P[1 .. m]$ and $T[s+1 .. s+m]$ are equal, the Rabin-Karp algorithm verifies the match with a brute force comparison.

This is still saving on comparisons over the brute force method, as we do not compare characters for failed hash matches: the strings cannot be the same.

The Rabin-Karp Algorithm

Here is the pseudocode:



```
RABIN-KARP-MATCHER(T, P, d, q)
1   n = T.length
2   m = P.length
3   h = d^{m-1} mod q
4   p = 0
5   t_0 = 0
6   for i = 1 to m                      // preprocessing
7       p = (dp + P[i]) mod q
8       t_0 = (dt_0 + T[i]) mod 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              print "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]) mod q
```

(The subscripts on t are just for exposition and can be ignored.)

- Line 3 initializes h to the value of the highest order digit that will be subtracted when shifting.
- Lines 4-8 compute the hash code p for P and the initial hash code t for the substring of T to be compared in each iteration (t will be updated by shifting digits as discussed).
- Like the brute-force algorithm, lines 9-14 iterate over all possible shifts s, but instead of comparing character by character, at first the hash codes are compared. If they match, brute force comparison is done.

You an see C code and a Java applet animation at http://www-igm.univ-mlv.fr/~lecroq/string/node5.html

Anaysis

Preprocessing is $\Theta(m)$: the loop 6-8 executes m times and with the modular arithmetic keeping the numbers within the size of one computer word the numerical work in the loop is O(1).

The worst case matching time is still $\Theta((n - m + 1)m)$, like brute force, because it is possible that every hash code matches and therefore every brute force comparison has to be done.

But this is highly unlikely in any realistic application (where we are searching for somewhat rare patterns in large texts). For constant number of valid shifts the text offers an argument that the expected time is O($n + m$), or O(n) since $m \le n$.

Extensions

A major advantage of Rabin-Karp is that it has natural extensions to other pattern matching problems, such as two-dimensional pattern matching (finding an m x m pattern in an array of n x n characters), or searching for multiple patterns at once. Therefore it remains an important pattern matching algorithm.

Dan Suthers
Last modified: Wed Apr 23 00:32:05 HST 2014
Images are from the instructor's material for Cormen et al. Introduction to Algorithms, Third Edition.