String Matching

Assume a text in an array $T[1, \ldots, n]$ and a pattern $P[1, \ldots, m]$, $m \leq n$. The characters are drawn from alphabet set $\Sigma$.

Goal is to find occurrences of $P$ in $T$.

Example. $T = \text{aabc baabaaa} \quad \Sigma = \{a, b, c\}$

$P = \text{aab}$

$P$ occurs at two places.

$P$ occurs with shift $s$ in $T$ if $0 \leq s \leq n-m$ and $T[s+1, \ldots, s+m] = P[1, \ldots, m]$.

Definitions for strings:

$\Sigma^*$: set of all strings possible with $\Sigma$

$\epsilon$: empty string, length zero, belongs to $\Sigma^*$

$|x|$: length of string $x$.

$xy$: concatenation of strings $x$ and $y$. 
Prefix: $\omega \in x$ is a prefix of $x$ if $x = \omega y$ for $y \in \Sigma^*$

Suffix: $\omega \in x$ is a suffix of $x$ if $x = y\omega$ for $y \in \Sigma^*$

The empty string is both prefix and suffix of every string.

\[
\begin{array}{c}
\text{aabc} \\
\text{prefix}
\end{array} \quad \begin{array}{c}
\text{dbba} \\
\text{suffix}
\end{array}
\]

$P_k$: $k$-character prefix of $P[1...m]$, that is, $P_k = P[1...k]$.

$T_k$: $k$-character prefix of $T[1...n]$.

String matching: Find all shifts $s \in [0, n-m]$ so that $P \sqsupseteq T_{s+m}$

Checking equality $x = y$: takes $O(t)$ time if $t$ is the length of the longest string that is prefix of both $x$ and $y$.

"aaabcaab = aabaabcaab" takes $3t+1$ checks.
A straightforward algorithm

Check if \( P[1...m] = T[s+1...s+m] \) for each \( s \in [0, n-m] \).

**Straight-Match (T, P)**

1. \( n := \text{length}[T] \)
2. \( m := \text{length}[P] \)
3. for \( s := 0 \) to \( n-m \)
   4. do if \( P[1...m] = T[s+1...s+m] \)
   5. then print "match with shifts."

Each check takes \( O(m) \) time and there are \( O(n-m+1) \) checks. So, total time \( O((n-m)m) \)

\[ T = \ldots ababaababa \quad P = abab \]

It should be obvious that starting from second position is redundant from looking at \( P \). This is utilized for efficiency later.
Rabin-Karp Algorithm

Here the strings are mapped to numbers which are matched instead. The algorithm has $\Theta(m)$ preprocessing time and $\Theta((n-m)m)$ running time. So, it is not better than the straightforward algorithm in the worst-case, but runs faster in practice.

Assume $\Sigma = \{0, 1, \ldots, d-1\}$. Each character is a digit in radix-$d$ notation.

Example: String 13456 in radix-10 is the number 13,456.

Let $p$ be value of $P$ in radix-10

t_s be value of $T[s+1, \ldots, s+m]$ in radix-10

Then $p = t_s$ iff $s$ is a valid shift, that is, $P$ matches in $T$ with shift $s$. 
Compute \( P \) from \( P \) in \( O(m) \) time
Compute all \( t_s \) from \( T \) in \( O(n-m+1) = O(n-m) \) time.
Check if \( p = t_s \) for all \( s \in [0, n-m] \).

This takes \( O(m) + O(n-m) = O(n) \) time.

One can compute \( p \) by Horner's rule.
\[
p = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + \cdots))
\]
to can be computed similarly from \( T \).
\[
t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]
\]

Example. \( m = 5 \), \( t_5 = 31415 \), \( T[s+1] = 3 \), \( T[s+5+1] = 2 \):
\[
t_{s+1} = 10(31415 - 10000 \cdot 3) + 2
= 14152
\]

So, after computing to in \( O(m) \) time,
we can compute all \( t_1, \ldots, t_{n-m} \) in
\( O(n-m) \) time.
The difficulty with the previous scheme is that $p$ and $t_s$ can become too large to assume that each operation (arithmetic) in constant-time computation.

Use modulo $q$ numbers: $p$, $t_s$ are all computed & modulo $q$, for some suitable $q$.

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

where $h = d^{m-1} \mod q$ \ldots \ [radix-d].

This solution keeps all numbers within a limited size $\leq q-1$, but $p = t_s$ check is no more perfect, since $t_s \equiv p \mod q$ does not imply $t_s = p$. However, if $t_s \not\equiv p \mod q$, then $t_s \not\equiv p$ for sure.

So, we can eliminate some invalid shifts quickly. But, if $t_s \equiv p \mod q$, then we have to check further.
\textbf{Rabin-Karp} (T, P, d, q)
\begin{align*}
n &:= \text{length} [T] \\
m &:= \text{length} [P] \\
h &:= d^{m-1} \pmod{q} \\
p &:= 0 \\
t_0 &:= 0 \\
\text{for } i &:= 1 \text{ to } m \\
&\quad \text{do } b := (dp + P[i]) \mod q \\
&\quad \quad \quad t_0 := (dt_0 + T[i]) \mod q \\
\text{for } s &:= 0 \text{ to } n - m \\
&\quad \text{do if } b = ts \\
&\quad \quad \text{then if } P[1 \ldots m] = T[s+1 \ldots s+m] \\
&\quad \quad \quad \text{then print "match with shifts"} \\
\text{update } ts &\quad \text{if } s < n - m \\
&\quad \quad \text{then } ts+1 := (d(ts - T[s+1])h) \\
&\quad \quad \quad \quad + T[s+m+1]) \mod q \\
\end{align*}

Because of the check at $\ast$, the worst-case time is $O((n-m)m)$. 
String Matching with FA

This method preprocesses the pattern P and builds a finite automaton. Then, matching needs $O(1)$ time per character in T. Thus, matching takes $O(n)$ time. But, preprocessing could be a little costly.

**FA:** $M = (Q, q_0, A, Z, S)$

- $Q$: finite states
- $q_0 \in Q$: start state
- $A \subseteq Q$: accepting states
- $Z$: input alphabet
- $S: Q \times Z \rightarrow Q$: a transition function.

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

If $M$ consumes a string and ends up in a final state, $M$ accepts the string. It rejects the string otherwise.
M induces a function $\phi : \Sigma^* \rightarrow Q$.

$\phi(\epsilon) = q_0$
$\phi(wo) = \delta(\phi(w), a)$ for $w \in \Sigma^*$, $a \in \Sigma$.

**Automaton for $P$.**

Define a suffix function $\sigma$ on $P$ as follows:

$\sigma(x) = \max \{k : P_k \triangleright x\}$ for $x \in \Sigma^*$.

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

**Ex.** $P = abab$

- $\sigma(ab) = 2$
- $\sigma(abaia) = 1$
- $\sigma(abbb) = 0$

$\sigma(x) = m$ iff $P \triangleright x$
For a pattern P[1...m] we define the automaton as:

- \( Q = \{0, 1, \ldots, m\} \), \( q_0 = \{0\} \), \( A = \{m\} \).
- \( \delta(q, a) = \sigma(P_q) \).

**Ex.** \( P = \text{ababaca} \)

\[
\begin{array}{cccc}
\text{State} & 0 & 1 & 2 & 3 \\
\hline
a & 1 & 0 & 0 & 0 \\
b & 1 & 1 & 2 & 0 \\
c & 2 & 3 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{State} & 4 & 5 & 6 & 7 \\
\hline
a & 1 & 4 & 6 & 0 \\
b & 5 & 0 & 0 & 0 \\
c & 7 & 0 & 0 & 0 \\
\end{array}
\]

\( i: 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \)

\( T[i]: \text{ababaca} \)

\( \phi(T[i]): 1 \ 2 \ 3 \ 4 \ 5 \ 4 \ 5 \ 6 \ 7 \ 2 \)

\( M \) is designed to maintain the following invariant: \( \phi(T[i]) = \sigma(T[i]) \).
Now, assuming that $M$ has been built for a $P$, we can write the algorithm for matching $P$ in $T$.

\textbf{Finite-Atomaton-match ($T, \delta, m$)}
\begin{align*}
n & := \text{length} (T) \\
q & := 0 \\
\text{for } i := 1 \text{ to } n \\
& \quad \text{do } q := \delta(q, T[i]) \\
& \quad \text{if } q = m \\
& \quad \text{then print "match with shift } i-m"
\end{align*}

\textbf{Lemma 1.} For any $x$ and $a \in \Sigma$, we have $\delta(xa) \leq \delta(x) + 1$.

\textbf{Proof.} Let $r = \delta(xa)$. If $r = 0$, then $r \leq \delta(x) + 1$ trivially true. So, assume $r \neq 0$.
\begin{align*}
P_r & \upharpoonright xa \text{ by def. of } \delta \\
P_{r-1} & \upharpoonright x \\
r-1 & \leq \delta(x).
\end{align*}
Lemma 2. If \( q = \sigma(x) \), then \( \sigma(xa) = \sigma(P_qa) \).

Proof.

1. \( P_q \uparrow x \) by def. of \( \sigma \)
2. \( P_qa \uparrow xa \) straightforward
3. \( r = \sigma(xa) \leq q + 1 \) by Lemma 1
4. \( P_r \uparrow xa \) by def. \( r = \sigma(xa) \).
5. \( |P_r| \leq |P_qa| \) by 2

1, 3, 4 \( \Rightarrow \) \( P_r \uparrow P_qa \Rightarrow r \leq \sigma(P_qa) \)
\( \Rightarrow \sigma(xa) \leq \sigma(P_qa) \)

We also have \( \sigma(P_qa) \leq \sigma(xa) \) by 1

Therefore, \( \sigma(xa) = \sigma(P_qa) \).

Theorem. If \( \phi \) is the final-state function, then \( \phi(T_i) = \sigma(T_i) \) \( \forall i \in [0, n] \).

Proof. By induction on \( i \).

For \( i = 0 \), trivially true since \( T_0 = \epsilon \Rightarrow \phi(T_0) = 0 = \sigma(T_0) \).
Assume \( \Phi(T_i) = \sigma(T_i) \) and prove \( \Phi(T_{i+1}) = \sigma(T_{i+1}) \).

Let \( q = \Phi(T_i) \) and \( a = T[i+1] \).

\[
\Phi(T_{i+1}) = \Phi(T_i a) \\
= \delta(\Phi(T_i), a) \\
= \delta(q, a) \\
= \sigma(P_{qa}) \quad \text{(Def. of } \delta \text{)} \\
= \sigma(T_i a) \quad \text{(Lemma 2)} \\
= \sigma(T_{i+1})
\]

By the Theorem, if \( M \) enters \( q \), then \( q \) is the largest value s.t. \( P_q \not\subseteq T_i \). Thus, \( q = m \) if and only if \( P \) has occurred just currently. So, the finite-automaton algorithm is correct.
Transition function

Transo Function \((P, \Sigma)\)

\[ m := \text{length}(P) \]

\[ \text{for } q := 0 \text{ to } m \]

\[ O(m|\Sigma|) \]

\[ \text{for each } a \in \Sigma \]

\[ k := \min(m+1, q+2) \]

\[ O(m) \]

\[ \text{repeat } k := k-1 \]

\[ O(m) \]

\[ \text{until } P_k \in P_a \]

\[ j(q, a) := k \]

\[ \text{return } \delta \]

Total complexity: \(O(m^3|\Sigma|)\).

It can be improved to \(O(m|\Sigma|)\).

Then FA approach takes \(O(n)\) matching time with \(O(m|\Sigma|)\) preprocessing time.