String Matching (Chapter 32)

- Text is an array $T[1..n]$, pattern is an array $P[1..m]$ and $m \leq n$
  - Elements of $P$ and $T$ are characters from alphabet $\Sigma$
  - Want to find where $P$ occurs in $T$
    + At what shifts
  - If $P$ occurs with shift $s$ in text $T$, we call $s$ a valid shift, otherwise $s$ is an invalid shift

String-Matching Problem: Find all valid shifts with which a pattern $P$ occurs in text $T$
Running Time

• Brute-force algorithm: ??

• Different approach:
  - Preprocess the pattern
  - Then do matching step

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Preprocessing time</th>
<th>Matching time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>0</td>
<td>$O((n - m + 1)m)$</td>
</tr>
<tr>
<td>Rabin-Karp</td>
<td>$\Theta(m)$</td>
<td>$O((n - m + 1)m)$</td>
</tr>
<tr>
<td>Finite automaton</td>
<td>$O(m</td>
<td>\Sigma</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>$\Theta(m)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

• $n$ and $m$ are not constants, as they can vary in size
Notation and Terminology

- **Strings**
  - $\Sigma$ set of symbols/characters
  - $\Sigma^*$ set of all finite length strings formed from characters in $\Sigma$
  - Zero-length string $\epsilon$ is in $\Sigma^*$
  - Length of string $x$ denoted by $|x|$
  - Concatenation of strings $x$ and $y$ denoted as $xy$

- What does it mean to say that $x = y$?
Notation and Terminology

• Strings
  - $\Sigma$ set of symbols/characters
  - $\Sigma^*$ set of all finite length strings formed from characters in $\Sigma$
  - Zero-length string $\epsilon$ is in $\Sigma^*$
  - Length of string $x$ denoted by $|x|$
  - Concatenation of strings $x$ and $y$ denoted as $xy$

• What does it mean to say that $x = y$?
  $x \in \Sigma^*$. So $x = x_1...x_n$ such that $n \geq 0$ and each $x_i \in \Sigma$
  $y \in \Sigma^*$. So $y = y_1...y_m$ such that $m \geq 0$ and each $y_i \in \Sigma$
  - We say that $x = y$ if $n = m$ and $x_i = y_i$ for $i \leq n$
Prefix and Suffix

Prefix:
w is prefix of x, denoted \( w \sqsubseteq x \), if there exists \( y \in \Sigma^* \) such that \( wy = x \)

Suffix:
w is suffix of x, denoted \( w \sqsupseteq x \) if there exists \( y \in \Sigma^* \) such that \( yw = x \)

• Examples
  \[
  \begin{align*}
  ab & \sqsubseteq abcca \\
  cca & \sqsubseteq abcca \\
  \epsilon & \sqsubseteq abcca
  \end{align*}
  \]

• For any strings \( x \) and \( y \) and any character \( a \), if \( x \sqsupseteq y \) then \( xa \sqsupseteq ya \)

• Suffix and Prefix:
  - reflexive?
  - symmetric?
  - transitive?
Lemma 32.1: Overlapping-suffix lemma
Suppose that $x$, $y$, and $z$ are strings such that $x \sqsubset z$ and $y \sqsubset z$.
If $|x| \leq |y|$, then $x \sqsubseteq y$.
If $|x| \geq |y|$, then $y \sqsubseteq x$.
If $|x| = |y|$ then $x = y$.

Proof:
(hand-waving)
Comparing Strings

• Comparing two equal-length strings
  - Might write this as $x == y$, but does not take constant time
  - Say that $z$ is longest prefix shared between $x$ and $y$ ($z \sqsubset x$ and $z \sqsubset y$)
  - If $|z| = t$, will take $\Theta(t + 1)$
    + Need to compare $t$ characters plus one more to find that strings are not equal
Overview

⇒ Naive String-Matching
• Rabin-Karp Algorithm
• String Matching with Finite Automata
Naive String-Matching Algorithm

\texttt{Naive-String-Matcher} \((T, P)\)

1. \(n = T.\text{length}\)
2. \(m = P.\text{length}\)
3. \textbf{for} \(s = 0\) \textbf{to} \(n - m\)
4. \quad \textbf{if} \(P[1..m] == T[s + 1..s + m]\)
5. \quad \text{print} \text{“Pattern occurs with shift”} \ s

- Running time \(O((n - m + 1)m)\)
  - Worst-case: text \(a^n\), pattern \(a^m\), must do \(O((n - m - 1)m)\) operations
  - No preprocessing, so running time is its matching time
- Naive: entirely ignores information gained about the text for one value of \(s\) when it considers other values of \(s\)
  - e.g., if \(P = \text{aaaa}\) and \(s=0\) is valid, then shifts of 1,2,3 are not valid

© P. Heeman, 2017  8 of 28  CS550  Class 15: Naive String-Matching
Questions from Textbook

32.1-2 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 $T$.

32.1-4 Suppose we allow the pattern $P$ to contain occurrences of a gap character $\diamond$ that can match an arbitrary string of characters (even one of zero length). For example $ab\diamond ba\diamond c$ occurs matches two ways in $cabccbacbabcabas$. Give a polynomial-time algorithm to determine whether such a pattern occurs in a given text $T$, and analyze the running time of your algorithm.
Overview

• Naive String-Matching
  ⇒ Rabin-Karp Algorithm
• String Matching with Finite Automata
Rabin-Karp Algorithm

• Uses $\Theta(m)$ preprocessing time

• Matching time: worst-case $\Theta((n - m + 1)m)$
  - Based on certain assumptions: average-case is better

• Makes use of elementary numeric notions
  - $a \mod c = b \mod c$

• Assume $\Sigma = \{0, 1, 2, ..., 9\}$: each char is a decimal digit
  - In general case, assume each char is a digit in base $d$ where $d = |\Sigma|$
  - Can view a $k$ length string as a $k$ length number

• Given a pattern $P[1..m]$, let $p$ be its corresponding decimal value

• For a text $T[1..n]$, let $t_s$ denote the decimal value of the length $m$
  substring $T[s+1..s+m]$ for $s = 0, 1, ..., n-m$
Computing \( p \) and \( t_s \)

- For example, assume \(|\Sigma| = 10\)
- Can compute \( p \) in time \( \Theta(m) \) using Horner’s rule
  
  \[
  p = \ldots((P[1]10 + P[2])10 + P[3])10 + \ldots + P[m - 1])10 + P[m]
  \]

- Similarly, we can compute \( t_0 \) from \( T[1..m] \) in time \( \Theta(m) \)
  
  \[
  t_0 = \ldots((T[1]10 + T[2])10 + T[3])10 + \ldots + T[m - 1])10 + T[m]
  \]

- All following \( t_s \) can be computed in \( \Theta(1) \) time
  
  \[
  + t_s \text{ based on } T[s + 1] \text{ down to } T[s + m]
  
  + t_{s+1}: \text{ subtract off highest digit } T[s + 1] \ast 10^{m-1}
  
  \text{ multiply rest by 10}
  
  \text{ add next digit } T[s + m + 1]
  \]
Computation

• Preprocessing:
  - compute $p$: $\Theta(m)$
  - compute $10^{m-1}$ (needed for computing $t_s$): $\Theta(m)$

• Compute all $t_0, t_1, \ldots, t_{n-m}$ in time $\Theta(m + (n - m))$

• But $t_s$ can be arbitrarily long (size $m$)
  - Computing $t_{s+1}$ needs $\Theta(m)$ time, not $\Theta(1)$
  - Comparing $p$ and $t_s$ needs $\Theta(m)$ time, not $\Theta(1)$
Doing Comparison in Constant Time

- Compute $p$ and $t_s$ in mod $q$
- Old way of computing $t_s$:
  \[ t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1] \]
  - Takes time $O(m)$ due to multiplication of $10^{m-1}$ and $T[s+1]$
- Facts about mod:
  \[
  (x + y) \mod q = ((x \mod q) + (y \mod q)) \mod q \\
  xy \mod q = (x + ... + x) \mod q \\
  = ((x \mod q) \ast y) \mod q
  \]
- New way: $t'_{s+1} = 10(t'_s - hT[s+1]) + T[s+m+1] \mod q$
  - $h = 10^{m-1} \mod q$
  - Now computation of $t_s$ done in size $q \ast 10$ not $m$
  - Pick $q$ so that $q \ast 10$ fits into a computer word and $q$ is prime
    + $q$ prime: helps make $t_s$ depend on whole substring
Spurious Hits

- $p = t_s \Rightarrow p \mod q = t_s \mod q$
- But not the other way around
- Testing $p' = t'_s$ will give false positives
- Anytime $p' = t'_s$, then need to check if $p = t_s$
  + This further check will take $O(m)$ time
- Hopefully spurious hits do not happen too often
  + Want $q$ as large as possible
RABIN-KARP-MATCHER(Τ, Ρ, d, q)

1. \( n = \text{length}(Τ) \)
2. \( m = \text{length}(Ρ) \)
3. \( h = d^{m-1} \mod q \)
4. \( p = 0 \)
5. \( t_0 = 0 \)
6. \( \text{for } i = 1 \text{ to } m \) \hspace{1em} // preprocessing
7. \( p = (dp + P[i]) \mod q \)
8. \( t_0 = (dt_0 + T[i]) \mod q \)
9. \( \text{for } s = 0 \text{ to } n - m \) \hspace{1em} // matching
10. \( \text{if } p == t_s \)
11. \( \hspace{1em} \text{if } P[1..m] == T[s + 1..s + m] \)
12. \( \hspace{2em} \text{print “Pattern occurs with shift” } s \)
13. \( \hspace{1em} \text{if } s < n - m \)
14. \( \hspace{2em} t_{s+1} = (d(t_s - T[s + 1]h) + T[s + m + 1]) \mod q \)

Doing this in base \( d \) where \( d = |\Sigma| \)
Overview

• Naive String-Matching
• Rabin-Karp Algorithm

⇒ String Matching with Finite Automata
String Matching with Finite Automata

- Faster yet: constant time per text character $O(n)$
  - Process text with a finite automata

- Finite Automata
  - 5 tuple
  
  + $Q$ is a finite set of states
  + $q_0 \in Q$ is the start state
  + $A \subseteq Q$ is a set of accepting states
  + $\Sigma$ is a finite input alphabet
  + $\delta : Q \times \Sigma \rightarrow Q$ is transition function

<table>
<thead>
<tr>
<th>state</th>
<th>input</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 0</td>
</tr>
<tr>
<td>1</td>
<td>0 0</td>
</tr>
</tbody>
</table>
• Accepting Strings
  - FA starts in state $q_0$
  - Reads characters of input string one at a time
  - If FA is in state $q$ and reads character $a$
    moves from state state $q$ to $\delta(q, a)$ (transitions)
  - Whenever FA is in $q \in A$, it has accepted string read so far
  - Otherwise it has rejected the string read so far

• FA $M$ induces a final-state function $\phi$
  - $\phi(w)$ maps to state $M$ is in at the end of reading $w$
  - Recursive Definition
    $+ \phi(\epsilon) = q_0$
    $+ \phi(wa) = \delta(\phi(w), a)$ for $w \in \Sigma^*, a \in \Sigma$
For any pattern $P$, it is possible to construct a FA

- FA is based on the $P$
- FA processes text
- FA is in an accepting state whenever the pattern is fully seen (at end of pattern in the text)
- FA monitors how much of pattern seen in processing text up to that point
  + Start point: seen nothing of pattern
  + A state for each successive character in pattern seen so far
  + State corresponding to all pattern characters seen is accept state
  + Transition from each state to its successive one
  + Back arcs when next character is not next in pattern
    + Need not be back to initial state since might be a smaller prefix of pattern that also matches so far

- Example: $P = \text{ababaca}$
Example: $P = \text{ababaca}$

(a)

(b)

(c)
Some More Notation

- For prefix $P[1..m]$, denote the first $k$ characters as $P_k$
  - So $P_0 = \epsilon$, and $P_m = P = P[1..m]$
- Similarly for text $T$, denote the first $k$ characters by $T_k$
- String matching problem can be written as:
  Find all shifts $s$ in range $0 \leq s \leq n-m$ such that $P \sqsupseteq T_{s+m}$
Suffix Function

• FA must track prefixes of $P$ that are a suffix of the text so far
  - If the longest suffix does not work, pursue next longest one

• Suffix function corresponding to $P \sigma : \Sigma^* \rightarrow \{0, 1, \ldots, m\}$
  - Length of the longest prefix of $P$ that is also a suffix of $x$
  - $\sigma(x) = \max\{k : P_k \sqsupseteq x\}$
  - Well defined since empty string $P_0 = \epsilon$ is a suffix of every string
  - Example: $P = ab$. $\sigma(ccaca) = ?? \quad \sigma(ccab) = ??$

• Implications of $\sigma$
  - If $P$ is of length $m$, $\sigma(x) = m$ iff $P \sqsupseteq x$
  - For $P$, if $x \sqsupseteq y$ then $\sigma(x) \leq \sigma(y)$
Defining the String Matching Automaton

- State set $Q$ is $\{0, 1, ..., m\}$
- Start state $q_0$ is 0. State $m$ is only accepting state
- Transition function $\delta$ is $\delta(q, a) = \sigma(P_qa)$
  - Makes sense!
  - We will be in the state that corresponds to the longest match of the prefix of $P$ with the suffix of the amount of string we have processed so far
  - But how do we prove this?
- Remember $\phi(w)$? Final state function
  - Outputs state that FA is in after processing $w$
  - Our definition of $\delta$ gives us $\phi(T_k) = \sigma(T_k)$
    + Just saying that the state that FA is in will match how much we are matching

© P. Heeman, 2017   24 of 28   CS550 Class 15: String Matching with Finite Automata
Reading the Next Character

• If FA is in state $q$ at $i$th character of $T$
  - $P_q$ is longest prefix of $P$ that is suffix of $T_i$
• FA then reads the next char $T[i + 1] = a$
  - Want to transition to state corresponding to longest prefix of $P$ that is a suffix of $T_i$ : $\sigma(T_i a)$
  - Since $P_q$ is the longest prefix of $P$ that is a suffix of $T_i$, $P_q$ captures everything that is important about $T_i$ (in terms of matching)
  - Longest suffix of $T_i a$ is also longest suffix of $P_q a$
  - So $\sigma(T_i a) = \sigma(P_q a)$
• Why is $\sigma(T_i a) = \sigma(P_q a)$ important?
  - Means we can compute $\sigma$ (and thus $\delta$) from just the all prefixes of the pattern and next possible characters
Example: \( P = ababaca \)

- **Forward arcs** capture next character matching
  - More and more of the prefix of \( P \) matches suffix of \( T_i \)
  - Example \( \delta(5, c) = 6 \)

- **Backward arcs**
  - When there is not a match
  - Example \( \delta(5, b) = 4 \)
    - Since in state 5, longest prefix of P that matches is \( P_5 = ababa \)
    - Next character is b. Longest prefix of P that matches \( P_5b = ababab \) is 4
Code: Matcher

FINITE-AUTOMATON-MATCHER \((T, \delta, m)\)

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

- Matcher runs in \(\Theta(n)\) time
  - Assuming \(\delta\) is just a table lookup
Computing Transition Function:

\[ \delta(q, a) = k \]

Running time: \( O(m^3|\Sigma|) \)
- Can actually do this in \( O(m|\Sigma|) \)