Strings and Pattern Matching

- Brute Force, Rabin-Karp, Knuth-Morris-Pratt
- Regular Expressions

What's up?

I'm looking for some string.

That's quite a trick considering that you have no eyes.

Oh yeah? Have you seen your writing? It looks like an EKG!
String Searching

• The previous slide is not a great example of what is meant by “String Searching.” Nor is it meant to ridicule people without eyes....

• The object of string searching is to find the location of a specific text pattern within a larger body of text (e.g., a sentence, a paragraph, a book, etc.).

• As with most algorithms, the main considerations for string searching are speed and efficiency.

• There are a number of string searching algorithms in existence today, but the three we shall review are Brute Force, Rabin-Karp, and Knuth-Morris-Pratt.
Brute Force

- The **Brute Force** algorithm compares the pattern to the text, one character at a time, until unmatching characters are found:

<table>
<thead>
<tr>
<th>TWO ROADS DIVERGED IN A YELLOW WOOD</th>
<th>ROADS</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWO ROADS DIVERGED IN A YELLOW WOOD</td>
<td></td>
</tr>
<tr>
<td>TWO ROADS DIVERGED IN A YELLOW WOOD</td>
<td></td>
</tr>
<tr>
<td>TWO ROADS DIVERGED IN A YELLOW WOOD</td>
<td></td>
</tr>
<tr>
<td>TWO ROADS DIVERGED IN A YELLOW WOOD</td>
<td></td>
</tr>
<tr>
<td>TWO ROADS DIVERGED IN A YELLOW WOOD</td>
<td></td>
</tr>
</tbody>
</table>

  - Compared characters are italicized.
  - Correct matches are in boldface type.

- The algorithm can be designed to stop on either the first occurrence of the pattern, or upon reaching the end of the text.
Brute Force Pseudo-Code

• Here’s the pseudo-code

\[
\begin{align*}
\text{do} & \\
\quad \text{if} \ (\text{text letter} == \text{pattern letter}) & \\
\qquad \text{compare next letter of pattern to next letter of text} & \\
\quad \text{else} & \\
\qquad \text{move pattern down text by one letter} & \\
\text{while} & \ (\text{entire pattern found} \ or \ \text{end of text})
\end{align*}
\]

cool cat Rolo went over the fence
__
cat
cool cat Rolo went over the fence
cat
cool cat Rolo went over the fence
cat
cool cat Rolo went over the fence
cat
cool__cat Rolo went over the fence
cat
cool cat Rolo went over the fence
cat
cool cat Rolo went over the fence
Brute Force-Complexity

• Given a pattern M characters in length, and a text N characters in length...

• **Worst case:** compares pattern to each substring of text of length M. For example, M=5.

• This kind of case can occur for image data.

1) `AAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH  5 comparisons made`

2) `AAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH  5 comparisons made`

3) `AAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH  5 comparisons made`

4) `AAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH  5 comparisons made`

5) `AAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH  5 comparisons made`

....

N) `AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   5 comparisons made
   AAAAH`

• Total number of comparisons: M (N-M+1)

• Worst case time complexity: O(MN)
Brute Force-Complexity (cont.)

• Given a pattern M characters in length, and a text N characters in length...

• **Best case if pattern found**: Finds pattern in first M positions of text. For example, M=5.

1) AAAAAA AAAAA AAAAA AAAAA AAAAA AAAAAA AAAAAAA
AAAAA 5 comparisons made

• Total number of comparisons: M

• Best case time complexity: O(M)
Brute Force-Complexity (cont.)

- Given a pattern M characters in length, and a text N characters in length...

- **Best case if pattern not found**: Always mismatch on first character. For example, M=5.

1) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   OOOOH 1 comparison made
2) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   OOOOH 1 comparison made
3) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   OOOOH 1 comparison made
4) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   OOOOH 1 comparison made
5) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   OOOOH 1 comparison made
...

N) AAAAAAAAAAAAAAAAAAAAAAAAHA
   1 comparison made OOOOH

- Total number of comparisons: N
- Best case time complexity: O(N)
Rabin-Karp

• The Rabin-Karp string searching algorithm calculates a **hash value** for the pattern, and for each M-character subsequence of text to be compared.

• If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence.

• If the hash values are equal, the algorithm will do a **Brute Force comparison** between the pattern and the M-character sequence.

• In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.

• Perhaps an example will clarify some things...
Rabin-Karp Example

Hash value of “AAAAA” is 37
Hash value of “AAAAAH” is 100

1) AAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAAH
   37≠100   1 comparison made
2) AAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAAH
   37≠100   1 comparison made
3) AAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAAH
   37≠100   1 comparison made
4) AAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAAH
   37≠100   1 comparison made

... 

N) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAAH
   5 comparisons made
   100=100
Rabin-Karp Algorithm

pattern is $M$ characters long

hash$_p$ = hash value of pattern
hash$_t$ = hash value of first $M$ letters in body of text

\[
\text{do} \\
\text{if} \ (\text{hash}_p == \text{hash}_t) \\
\quad \text{brute force comparison of pattern and selected section of text} \\
\quad \text{hash}_t = \text{hash value of next section of text, one character over} \\
\text{while (end of text or brute force comparison == true)}
\]
Rabin-Karp

• Common Rabin-Karp questions:

  “What is the hash function used to calculate values for character sequences?”

  “Isn’t it time consuming to hash every one of the M-character sequences in the text body?”

  “Is this going to be on the final?”

• To answer some of these questions, we’ll have to get mathematical.
Rabin-Karp Math

- Consider an M-character sequence as an M-digit number in base $b$, where $b$ is the number of letters in the alphabet. The text subsequence $t[i..i+M-1]$ is mapped to the number

$$x(i) = t[i] \cdot b^{M-1} + t[i+1] \cdot b^{M-2} + \ldots + t[i+M-1]$$

- Furthermore, given $x(i)$ we can compute $x(i+1)$ for the next subsequence $t[i+1..i+M]$ in constant time, as follows:

$$x(i+1) = t[i+1] \cdot b^{M-1} + t[i+2] \cdot b^{M-2} + \ldots + t[i+M]$$

$$x(i+1) = x(i) \cdot b$$

Shift left one digit

$$- t[i] \cdot b^M$$

Subtract leftmost digit

$$+ t[i+M]$$

Add new rightmost digit

- In this way, we never explicitly compute a new value. We simply adjust the existing value as we move over one character.
Rabin-Karp Math Example

- Let’s say that our alphabet consists of 10 letters.
- our alphabet = a, b, c, d, e, f, g, h, i, j
- Let’s say that “a” corresponds to 1, “b” corresponds to 2 and so on.

  The hash value for string “cah” would be ...

  \[3 \times 100 + 1 \times 10 + 8 \times 1 = 318\]
Rabin-Karp Mods

- If $M$ is large, then the resulting value ($\sim b^M$) will be enormous. For this reason, we hash the value by taking it \texttt{mod} a prime number $q$.

- The \texttt{mod} function (\% in Java) is particularly useful in this case due to several of its inherent properties:
  - $[(x \mod q) + (y \mod q)] \mod q = (x+y) \mod q$
  - $(x \mod q) \mod q = x \mod q$

- For these reasons:

  $$h(i) = ((t[i] \cdot b^{M-1} \mod q) + (t[i+1] \cdot b^{M-2} \mod q) + \ldots + (t[i+M-1] \mod q)) \mod q$$

  $$h(i+1) = (h(i) \cdot b \mod q)$$

  \hspace{20pt} \text{Shift left one digit}

  $$-t[i] \cdot b^M \mod q$$

  \hspace{20pt} \text{Subtract leftmost digit}

  $$+t[i+M] \mod q$$

  \hspace{20pt} \text{Add new rightmost digit}

  \hspace{10pt} \mod q$$
Rabin-Karp Complexity

- If a sufficiently large prime number is used for the hash function, the hashed values of two different patterns will usually be distinct.

- If this is the case, searching takes $O(N)$ time, where $N$ is the number of characters in the larger body of text.

- It is always possible to construct a scenario with a worst case complexity of $O(MN)$. This, however, is likely to happen only if the prime number used for hashing is small.
The Knuth-Morris-Pratt Algorithm

• The Knuth-Morris-Pratt (KMP) string searching algorithm differs from the brute-force algorithm by keeping track of information gained from previous comparisons.

• A failure function \((f)\) is computed that indicates how much of the last comparison can be reused if it fails.

• Specifically, \(f\) is defined to be the longest prefix of the pattern \(P[0,...,j]\) that is also a suffix of \(P[1,...,j]\)
  - Note: not a suffix of \(P[0,...,j]\)

• Example:
  - value of the KMP failure function:

<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P[j])</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>(f(j))</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

• This shows how much of the beginning of the string matches up to the portion immediately preceding a failed comparison.
  - if the comparison fails at (4), we know the a,b in positions 2,3 is identical to positions 0,1
The KMP Algorithm (contd.)

• the KMP string matching algorithm: Pseudo-Code

Algorithm KMPMatch($T,P$)

**Input:** Strings $T$ (text) with $n$ characters and $P$ (pattern) with $m$ characters.

**Output:** Starting index of the first substring of $T$ matching $P$, or an indication that $P$ is not a substring of $T$.

\[
\begin{align*}
f & \leftarrow \text{KMPFailureFunction}(P) \quad \{\text{build failure function}\} \\
i & \leftarrow 0 \\
j & \leftarrow 0 \\
\text{while } i & < n \text{ do} \\
\text{if } P[j] & = T[i] \text{ then} \\
\quad & \text{if } j = m - 1 \text{ then} \\
\quad & \quad \text{return } i - m - 1 \quad \{\text{a match}\} \\
\quad & \quad i \leftarrow i + 1 \\
\quad & \quad j \leftarrow j + 1 \\
\text{else if } j & > 0 \text{ then} \quad \{\text{no match, but we have advanced}\} \\
\quad & \quad j \leftarrow f(j-1) \quad \{j indexes just after matching prefix in P\} \\
\text{else} & \\
\quad i & \leftarrow i + 1 \\
\text{return} & \text{“There is no substring of } T \text{ matching } P”
\end{align*}
\]
The KMP Algorithm (contd.)

• The KMP failure function: Pseudo-Code

Algorithm KMPFailureFunction($P$);

Input: String $P$ (pattern) with $m$ characters

Output: The failure function $f$ for $P$, which maps $j$ to the length of the longest prefix of $P$ that is a suffix of $P[1,\ldots,j]$

\[
i \leftarrow 1
\]
\[
j \leftarrow 0
\]

while $i \leq m-1$ do
    if $P[j] = T[j]$ then
        {we have matched $j + 1$ characters}
        \[
f(i) \leftarrow j + 1
\]
        \[
i \leftarrow i + 1
\]
        \[
j \leftarrow j + 1
\]
    else if $j > 0$ then
        {j indexes just after a prefix of $P$ that matches}
        \[
j \leftarrow f(j-1)
\]
    else
        {there is no match}
        \[
f(i) \leftarrow 0
\]
        \[
i \leftarrow i + 1
\]
The KMP Algorithm (contd.)

- A graphical representation of the KMP string searching algorithm

```
abacabacabacaba
1 2 3 4 5 6
abacab

abacab
7

abacab
8 9 10 11 12
abacab

abacab
13

abacab
14 15 16 17 18 19
abacab
```

no comparison needed here
The KMP Algorithm (contd.)

• Time Complexity Analysis
• define $k = i - j$

• In every iteration through the while loop, one of three things happens.
  - 1) if $T[i] = P[j]$, then $i$ increases by 1, as does $j$
      $k$ remains the same.
  - 2) if $T[i] \neq P[j]$ and $j > 0$, then $i$ does not change
      and $k$ increases by at least 1, since $k$ changes
      from $i - j$ to $i - f(j-1)$
  - 3) if $T[i] \neq P[j]$ and $j = 0$, then $i$ increases by 1 and
      $k$ increases by 1 since $j$ remains the same.

• Thus, each time through the loop, either $i$ or $k$
  increases by at least 1, so the greatest possible
  number of loops is $2n$

• This of course assumes that $f$ has already been
  computed.

• However, $f$ is computed in much the same manner as
  KMPMatch so the time complexity argument is
  analogous. KMPFailureFunction is $O(m)$

• Total Time Complexity: $O(n + m)$
Regular Expressions

• notation for describing a set of strings, possibly of infinite size

• \( \varepsilon \) denotes the empty string

• \( ab + c \) denotes the set \( \{ab, c\} \)

• \( a^* \) denotes the set \( \{\varepsilon, a, aa, aaa, ...\} \)

• Examples
  - \( (a+b)^* \) all the strings from the alphabet \( \{a,b\} \)
  - \( b^*(ab^a)^*b^* \) strings with an even number of a’s
  - \( (a+b)^*\text{sun}(a+b)^* \) strings containing the pattern “sun”
  - \( (a+b)(a+b)(a+b)a \) 4-letter strings ending in a