Seminumerical String Matching

CMSC 701
Semi-numerical string matching:

Instead of focusing on comparing characters, think of string as a sequence of bits or numbers and use arithmetic operations to search for patterns.

Two algorithms:
- Rabin-Karp
- Shift-And

Both tend to be better for short patterns.
Rabin-Karp
(Following CLR Chapter 34)
Characters as digits

- Assume $\Sigma = \{0, \ldots, 9\}$
- Then a string can be thought of as the decimal representation of a number:

$$427328$$

- In general, if $|\Sigma| = d$, a string represents a number in base $d$.
- Let $p =$ the number represented by query $P$.
- Let $t_s =$ the number represented by the $|P|$ digits of $T$ that start at position $s$.

$P$ occurs at position $s$ of $T \iff p = t_s$. 
Computing $p$ and $t_s$

- Use Horner’s rule to compute $p$ in time $O(|P|=m)$:

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

- Example: $427328 = (8 + 10(2 + 10(3 + 10(7 + 10(2 + 10 \times 4))))))$

  “Left shift” by 1 digit

- $t_0$ can be computed the same way in time $O(|P|=m)$.

- $t_s$ can be computed from $t_{s-1}$ in $O(1)$ time:

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

  - shift left by 1 digit
  - remove high-order digit
  - add next digit of $T$ as the low-order digit
Rabin-Karp

Problem: \( p \) and \( t_s \) might be huge numbers.

Solution: compute everything modulo some prime \( q \).

- If \( 10q \) is \( \leq \) word size, then \( p \mod q \) and \( t_s \mod q \) can be computed in a single word.

- If \( p \) occurs at \( t_s \), then \( p \equiv t_s \mod q \)

New problem: If \( p \equiv t_s \mod q \), it doesn’t necessarily mean there is a match at \( s \).

New solution: if \( p \equiv t_s \mod q \), check match explicitly.

Worst-case runtime = \( O(mn) \), if every position is a match or false positive.
Shift-And
(Following Gusfield Chapter 4)
**Shift-And Algorithm**

\[ M[i,j] := 1 \text{ iff prefix } i \text{ of } P \text{ matches a substring of } T \text{ ending at } j: \]

\[ M[i,j] = P[i] = T[j] \text{ and } P[1..i-1] = T[\text{ending @ } j-1] \]

1’s in last row will indicate where \( P \) matches \( T \).
Computing $M$ by columns

\[ M[i,j] = P[i] = T[j] \text{ and } P[1..i-1] = T[\text{ending @ } j-1] \]

\[ M[i-1,j-1] \]

**Def.** $U_P(x) = |P|$-bit vector where $i^{th}$ entry is 1 if $P[i] = x$, 0 otherwise.

Compute columns of $M$ left to right:

\[ M[\cdot,j] = U_P(T[j]) \& (1; M[\cdot,j-1]) \]

- $j^{th}$ column of $M$
- 1 where $P[i] = T[j]$
- previous column of $M$, shifted down by 1 (prepended with a 1)

\[
\begin{array}{cccc}
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 \\
\end{array}
\]

first entry always 1 because red condition is empty for $i = 1$. 
• Only the current and previous columns of $M$ are needed, so space is $O(|P|)$.

• Worst case running time $O(|P| \times |T|)$.

• But if $|P|$ in bits $\leq$ computer word, each column of $M$ can be computed in constant time, leading to an $O(|T|)$ algorithm.
Extension to approximate matching

\[ M^l[i,j] = i^{\text{th}} \text{ prefix of } P \]
matches suffix ending at \( j \) of \( T \) →
with \( \leq l \) mismatches.

\[ M[l][j] = M[l-1][j] \text{ or } \left( \text{bs}(M[l](j-1)) \text{ and } U(T(j)) \right) \text{ or } \text{bs}(M[l-1][j-1]) \]

\( i-1 \) characters of \( P \) match
with \( \leq l \) mismatches and
\( j^{\text{th}} \) character matches.

bs(\( v \)) := (1; \( v \)) truncated to \( n \) dimensions.
Seminumerical Matching

Often effective when pattern is small.

Asymptotically, not the best run time, but if operations can be done fast in hardware, these algorithms can be good choices.

Also, provide a different perspective on the string matching problem.