String Matching: Rabin-Karp Algorithm

Greg Plaxton
Theory in Programming Practice, Fall 2005
Department of Computer Science
University of Texas at Austin
The (Exact) String Matching Problem

• The (exact) string matching problem: Given a text string $t$ and a pattern string $p$, find all occurrences of $p$ in $t$

• A naive algorithm for this problem simply considers all possible starting positions $i$ of a matching string within $t$, and compares $p$ to the substring of $t$ beginning at each such position $i$
  – The worst-case complexity of this algorithm is $\Theta(mn)$, where $m$ denotes the length of $p$ and $n$ denotes the length of $t$
  – Can we do better?
Three Efficient String Matching Algorithms

• Rabin-Karp (today)
  – This is a simple randomized algorithm that tends to run in linear time in most scenarios of practical interest
  – The worst case running time is as bad as that of the naive algorithm, i.e., $\Theta(mn)$

• Knuth-Morris-Pratt
  – The worst case running time of this algorithm is linear, i.e., $O(m+n)$

• Boyer-Moore
  – This algorithm tends to have the best performance in practice, as it often runs in sublinear time
  – The worst case running time is as bad as that of the naive algorithm
The Rabin-Karp String Matching Algorithm

• Assume the text string $t$ is of length $m$ and the pattern string $p$ is of length $n$

• Let $s_i$ denote the length-$n$ contiguous substring of $t$ beginning at offset $i \geq 0$
  – So, for example, $s_0$ is the length-$n$ prefix of $t$

• The main idea is to use a hash function $h$ to map each $s_i$ to a good-sized set such as the set of the first $k$ nonnegative integers, for some suitable $k$
  – Initially, we compute $h(p)$
  – Whenever we encounter an $i$ for which $h(s_i) = h(p)$, we check for a match as in the naive algorithm
  – If $h(s_i) \neq h(p)$, we don’t need to check for a match
The Choice of Hash Function

• It should be easy to compare two hash values
  – For example, if the range of the hash function is a set of sufficiently small nonnegative integers, then two hash values can be compared with a single machine instruction

• The number of false positives induced by the hash function should be similar to that achieved by a “random” function
  – If the range of the hash function is of size $k$, we’d like each hash value to be achieved by approximately the same number of $n$-symbol strings (where $n$ is the length of the pattern)

• It should be easy (e.g., a constant number of machine instructions) to compute $h(s_{i+1})$ given $h(s_i)$
A Possible Choice for the Hash Function

- Suppose we hash each string to the XOR of the ASCII values of its characters
  - Is this a good choice of hash function with respect to the criteria mentioned on the previous slide?

- What if we hash each string to the sum of the ASCII values of its characters?

- What if we view each string as a nonnegative number?
  - For example, an ASCII string may be viewed as a base 256 number
  - Alternatively, an $n$-symbol ASCII string may be viewed as an $(8n)$-bit number
A Good Choice for the Hash Function

• View each string as a nonnegative number, but take the result modulo $k$ for some suitable modulus $k$

• For example, we might take $k$ to be $2^{32}$, to ensure that the hash values can be stored in a 32-bit integer

• In practice the modulus $k$ is generally taken to be a prime (e.g., a 32-bit prime) in order to better destroy any structure in the input data
  – For example, note that the 8-bit ASCII codes for printable characters all begin with a 0
  – So if we use $k = 2^{32}$, bits 7, 15, 23, and 31 of the hash of a printable string are guaranteed to be zero

• But can we still compute $h(s_{i+1})$ from $h(s_i)$ efficiently?