String-Matching

Dr. G. P. Gupta

Outline

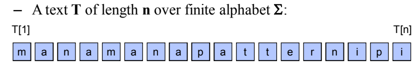
- **The Naive algorithm**
  - How would you do it?
- **The Rabin-Karp algorithm**
  - Ingenious use of primes and number theory
  - In practice optimal
- **String Matching with finite Automata**
- **The Knuth-Morris-Pratt algorithm**
  - Skip patterns that do not match
  - This is optimal

String Matching : **Applications**

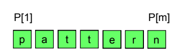
- Problem arises obviously in text-editing programs
- Efficient algorithms for this problem can greatly aid the responsiveness of the text editing program
- Other applications include:
  - Detecting plagiarism
  - Bioinformatics
  - Data Compression

String Search Task

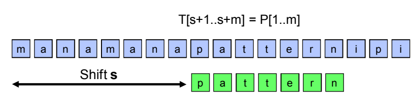
- Given
  - A text **T** of length **n** over finite alphabet **Σ**:



  - A pattern **P** of length **m** over finite alphabet **Σ**:



- Output
  - All occurances of **P** in **T**



Problem Definition

- **String-Matching Problem**:
  - Given a text string T and a pattern string P, ***find all valid shifts*** with which a given pattern P occurs in a given text T.

String-matching problem

Given:

- Text $T[1\,..\,n]$
- Pattern $P[1\,..\,m]$, where $m \leq n$

Characters of text and pattern are drawn from a common finite alphabet Σ: $T \in \Sigma^*$ and $P \in \Sigma^*$.

Find:
All occurrences of pattern P in T, that is, all valid shifts s, where $0 \leq s \leq n-m$, such that

$$T[s+1\,..\,s+m] = P[1\,..\,m]$$

or

$$T[s+j] = P[j], \quad j = 1, \ldots, m$$

String Matching Algorithms

- **Naive Algorithm**
  - Worst-case running time in O((n-m+1) m)
- **Rabin-Karp**
  - Worst-case running time in O((n-m+1) m)
  - Better than this on average and in practice
- **Finite Automaton-Based**
  - Worst-case running time in O(n + m|S|)
- **Knuth-Morris-Pratt**
  - Worst-case running time in O(n + m)

Notation & Terminology

- S* = set of all finite-length strings formed using characters from alphabet S
- Empty string: e
- |x| = length of string x
- w is a prefix of x: w x ab abcca
- w is a suffix of x: w x cca abcca
- prefix, suffix are *transitive*

Overlapping Suffix Lemma

Lemma 32.1 (Overlapping-suffix lemma)
Suppose that x, y, and z are strings such that $x \sqsupset z$ and $y \sqsupset z$. If $|x| \le |y|$, then $x \sqsupset y$. If $|x| \ge |y|$, then $y \sqsupset x$. If $|x| = |y|$, then $x = y$.

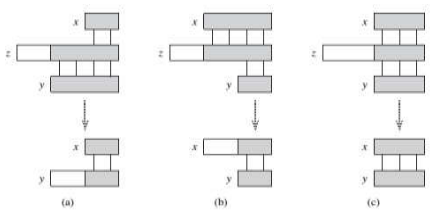


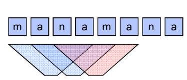
Figure 32.3 A graphical proof of Lemma 32.1. We suppose that $x \sqsupset z$ and $y \sqsupset z$. The three parts of the figure illustrate the three cases of the lemma. Vertical lines connect matching regions (shown shaded) of the strings. **(a)** If $|x| \le |y|$, then $x \sqsupset y$. **(b)** If $|x| \ge |y|$, then $y \sqsupset x$. **(c)** If $|x| = |y|$, then $x = y$.

String Matching Algorithms

Naive Algorithm

Naive Algorithm

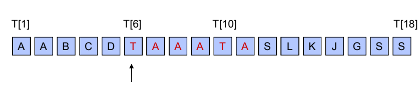
```
Naive-String-Matcher(T,P)
Ø  n ← length(T)
Ø  m ← length(P)
Ø  for s ← 0 to n-m do
Ø       if P[1..m] = T[s+1 .. s+m] then
Ø               return "Pattern occurs with shift s"
Ø
```



- **Fact:**
  - The naive string matcher needs **worst-case** running time **O((n-m+1) m)**
  - For n = 2m this is **O(n²)**
  - The naive string matcher is not optimal, since string matching can be done in time **O(m + n)**

Can we do better than O(nm)?

- **Shifting the text string over by 1 every time can be wasteful**
- Example:
  - Suppose that the target pattern is "**TAAATA**" and source string is "**AABCDTAAATASLKJGSSK**"



  - If there is a match starting at position 6 of the text string, then position 7 can't possibly work; in fact, **the next feasible starting position would be 10 in this case**

- **This suggests that there may be ways to speed up string matching**

String Matching Algorithms

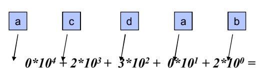
Rabin-Karp

Rabin-Karp Algorithm
Coding Strings as Numbers

- Assume each character is digit in radix-d notation (e.g. d=10)
- The string $d_n \ldots d_0$ represents the number
  - $\mathbf{d_n*10^n + \ldots + d_1*10^1 + d_0*10^0}$
  - **"324" = $\mathbf{3*10^2 + 2*10^1 + 4*10^0}$**
- Given any "**alphabet**" of possible "**digits**" (like 0...9 in the case of decimal notation), we can associate a number with a string of symbols from that alphabet
  - Let ***d*** be the total number of symbols in the alphabet
  - Order the symbols in some way, as ***a(0)...a(d-1)***
  - Associate to each symbol ***a(k)*** the value ***k***
  - Then view each string ***w[1...n]*** over this alphabet as corresponding to a number in "**base d**"

Rabin-Karp Algorithm
Coding Strings as Numbers

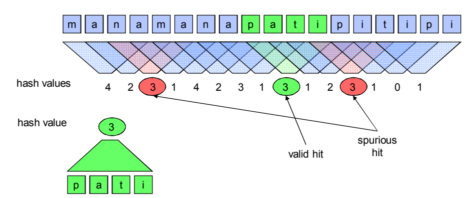
- Example, for the alphabet Σ = ***{a, b, c, d, e, f, g, h, i, j}***, *where* $|\Sigma|$ = 10 Interpret it as ***{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.***
  - Value of the string "***acdab***" would be:



$0*10^4 + 2*10^3 + 3*10^2 + 0*10^1 + 2*10^0$ = 02302.

Rabin-Karp Algorithm

- **Idea** – **Compute:**
  - **hash value for pattern P and**
  - **hash value for each sub-string of T of length m**



Rabin-Karp Algorithm
Compute Hash value using Horner's Rule

- Compute

$$S_m(P) = \sum_{i=1}^{m} d^{m-i} P[i] \bmod q$$

- by using

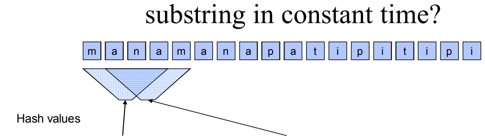
$$S_m(P) \equiv \sum_{i=1}^{m} d^{m-i} P[i] \equiv d\left(\sum_{i=1}^{m-1} d^{m-i-1} P[i]\right) + P[m] \equiv dS_{m-1}(P[1..m-1]) + P[m] \pmod q$$

  - Run time is **O(m)**

- Example
  - Then d = 10, q = 13
  - Let P = 0815

$S_4(0815) = ((((\mathbf{0} \cdot 10 + \mathbf{8}) \cdot 10) + \mathbf{1}) \cdot 10) + \mathbf{5} \bmod 13 =$
$(((\mathbf{8} \cdot 10) + \mathbf{1}) \cdot 10) + \mathbf{5} \bmod 13 =$
$(3 \cdot 10) + \mathbf{5} \bmod 13 = 9$

Rabin-Karp Algorithm
How to compute the hash value of the next substring in constant time?

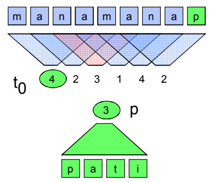


S_m(T[1..m]) = 31415 **S_m(T[2..m+1]) = 14152**

$$S_m(T[2..m+1]) \equiv d(S_m(T[1..m]) - d^{m-1}T[1]) + T[m+1] \pmod q$$

- Ex. d = 10
  - T_{s+1} = 10(**31415** – 10000 * 3) + 2 = **14152**
- Computation of hash value S_m(T[2..m + 1]) can be performed in constant time from hash value S_m(T[1..m])

Rabin-Karp Algorithm



Rabin-Karp-Matcher(T,P,d,q)

```
n ← length(T)
m ← length(P)
h ← d^(m-1) mod q
p ← 0
t0 ← 0
for i ← 1 to m do
   p ← (d p + P[i]) mod q
   t0 ← (d t0 + T[i]) mod q
od
for s ← 0 to n-m do
   if p = ts then
      if P[1..m] = T[s+1..s+m] then return "Pattern occurs with shift" s
   fi
   if s < n-m then
      ts+1 ← (d(ts-T[s+1]h) + T[s+m+1]) mod q
   fi
od
```

Hash value match
Now test for
false positive

Update hash value for
T[s+1..s+m] using
hash value of T[s..s+m-1]

Rabin-Karp Algorithm Correctness

- *<pre-condition>*: T is a string of characters over an alphabet of size d, P is string of characters over an alphabet of size d and |P| <= |T|, d is the size of the alphabet and q is a prime number
- *<post-condition>*: Print all valid shifts s, where 0 <= s <= |*T*|, with which a given pattern *P* occurs in a given text *T*

```
p ← 0
t0 ← 0
for i ← 1 to m do
   p ← (d p + P[i]) mod q
   t0 ← (d t0 + T[i]) mod q
od
```

Establishing Loop Invariant:
Lines 1-6 establish the loop invariant

<exit-condition>: s = n - m + 1

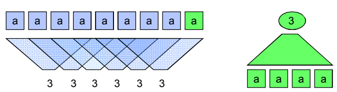
```
for s ← 0 to n-m do
   if p = ts then
      if P[1..m] = T[s+1..s+m]
then return "Pattern occurs with
shift" s
   fi
   if s < n-m then
      ts+1 ← (d(ts-T[s+1]h) +
T[s+m+1]) mod q
   fi
od
```

Maintaining Loop Invariant:
Lines 3-5 maintain the validity of the loop invariant.
Line 3 verifies that each hash value match is a valid shift. Therefore in each iteration of the loop, we only print valid shifts of P in T.

Loop Invariant:
Whenever line 2 is executed,
t_s = T[s + 1..s + m] mod q and we have printed all valid shifts for values strictly smaller than s

Rabin-Karp Algorithm
Performance Analysis

- The worst-case running time of the Rabin-Karp algorithm is **O(m (n-m+1))**
  - Example: $\mathbf{P = a^m}$ and $\mathbf{T = a^n}$, since each of the [n-m+1] possible shifts is valid



- Probabilistic analysis
  - The probability of a false positive hit for a random input is **1/q**
  - The expected number of false positive hits is **O(n/q)**
  - The expected run time of Rabin-Karp is **O(n) + O(m (v + n/q))** if **v** is the number of valid shifts (hits)
- If we choose **q ≥ m** and have only a constant number of hits, then the expected run time of Rabin-Karp is **O(n + m)**.

Rabin-Karp Algorithm
Review

- **Idea**: Converts strings into decimal numbers
- Perform preprocessing of substrings to skip over false matches
- Worst-case running time O(n*m) because of the need to validate a match
- In practice, the Rabin-Karp algorithm runs in O(n) which turns out to be optimal

String Matching with Finite Automata

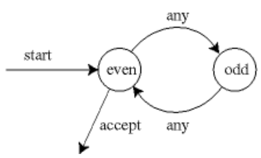
Finite Automata

A finite automaton is a quintuple (Q, Σ, δ, s, F):

- Q: the finite **set of states**
- Σ: the finite **input alphabet**
- δ: the "transition function" from QxΣ to Q
- s ∈ Q: the start state
- F ⊂ Q: the set of final (accepting) states

How it works

A finite automaton accepts strings in a specific language. It begins in state q_0 and reads characters one at a time from the input string. It makes transitions (ϕ) based on these characters, and if when it reaches the end of the tape it is in one of the accept states, that string is accepted by the language.



http://www.ics.uci.edu/~eppstein/161/960222.html

The Suffix Function

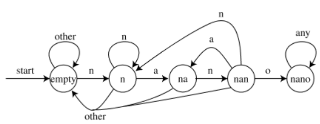
In order to properly search for the string, the program must define a **suffix function (σ)** which checks to see how much of what it is reading matches the search string at any given moment.

$$\sigma(x) = \max\{k : P_k \sqsupset x\}$$

P = abaabc
P_1 = a
P_2 = ab
P_3 = aba
P_4 = abaa
σ(abbaba) = aba

http://www.cs.duke.edu/education/courses/cps130/fall98/lectures/lect14/node31.html

Example: nano



	n	a	o	other
empty:	n	ε	ε	ε
n:	n	na	ε	ε
na:	nan	ε	ε	ε
nan:	n	na	nano	ε
nano:	nano	nano	nano	nano

String-Matching Automata

- For any pattern P of length m, we can define its string matching automata:
  - $Q = \{0,\dots,m\}$ (states)
  - $q_0 = 0$ (start state)
  - $F = \{m\}$ (accepting state)
  - $\delta(q,a) = \sigma(P_q a)$

The transition function chooses the next state to maintain the invariant:

$$\phi(T_i) = \sigma(T_i)$$

After scanning in i characters, the state number is the longest prefix of P that is also a suffix of T_i.

Finite-Automaton-Matcher

The simple loop structure implies a running time for a string of length *n* is O(n).

However: this is only the running time for the actual string matching. It does not include the time it takes to compute the transition function.

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
n ← length[T]
q ← 0
for i ← 1 to n
do q ← δ(q, T[i])
   if q = m
      then s ← i − m
           print "Pattern occurs at shift" s
```

Computing the Transition Function

Compute-Transition-Function (P, Σ)

$m \leftarrow$ length[P]

For $q \leftarrow 0$ to m

  do for each character $a \in \Sigma$

    do $k \leftarrow \min(m+1, q+2)$

      repeat $k \leftarrow k-1$

        until $P_k \supset P_q a$

      $\delta(q,a) \leftarrow k$

return δ

This procedure computes $\delta(q,a)$ according to its definition. The loop on line 2 cycles through all the states, while the nested loop on line 3 cycles through the alphabet. Thus all state-character combinations are accounted for. Lines 4-7 set $\delta(q,a)$ to be the largest k such that $P_k \supset P_q a$.

Running Time of Compute-Transition-Function

Running Time: $O(m^3 |\Sigma|)$

- Outer loop: $m |\Sigma|$
- Inner loop: runs at most $m+1$
- $P_k \supset P_q a$: requires up to m comparisons

References

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 32.2: The Rabin-Karp algorithm, pp.911–916.
- Karp, Richard M.; Rabin, Michael O. (March 1987). "Efficient randomized pattern-matching algorithms". IBM Journal of Research and Development 31 (2), 249-260.
- String Matching: Rabin-Karp Algorithm www.cs.utexas.edu/users/plaxton/c/337/05f/slides/StringMatching-1.pdf