String Matching

Algorithm Design and Analysis
(Week 7)

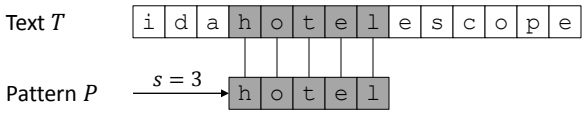
Battle Plan

- String matching problem
- Notation and terminology
- Four different algorithms

Algorithm	Preprocessing Time	Matching Time
Naïve	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite automaton	$O(m\|\Sigma\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

String-Matching Problem

- "Where's the `hotel` in `idahotelescope`?"
- Formalization of the **string-matching problem**
  - **Text** is an array $T[1..n]$ of length n
  - **Pattern** is an array $P[1..m]$ of length $m \leq n$
  - T and P are drawn from a finite alphabet Σ, they are often called **strings** of characters
  - P **occurs with shift** s in T if $0 \leq s \leq n - m$ and $T[s + 1..s + m] = P[1..m]$



Notation and Terminology

- Strings
  - Σ^* set of all finite-length strings with characters from Σ
  - ϵ zero-length **empty string** also belongs to Σ^*
  - $|x|$ length of a string x
  - xy concatenation of strings x and y has length $|x| + |y|$
- Prefix and suffix
  - string w is a **prefix** of a string x, denoted as $w \sqsubset x$, if $x = wy$ for some string $y \in \Sigma^*$
  - string w is a **suffix** of a string x, denoted as $w \sqsupset x$, if $x = yw$ for some string $y \in \Sigma^*$
  - S_k denotes the k-character prefix $S[1..k]$ of the string $S[1..n]$ and thus $S_0 = \epsilon$ and $S_n = S = S[1..n]$.

Observations

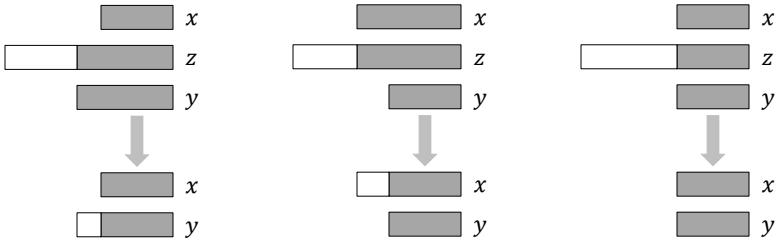
- Strings
  - $|\epsilon| = 0$
- Prefix and suffix
  - for any string x, $\epsilon \sqsubset x$ and $\epsilon \sqsupset x$
  - if $w \sqsubset x$ or $w \sqsupset x$, then $|w| \leq |x|$
  - for any two strings x and y and any character a, $x \sqsubset y \rightarrow ax \sqsubset ay$ and $x \sqsupset y \rightarrow xa \sqsupset ya$
  - both $\sqsubset$ and $\sqsupset$ are transitive relations
- Reformulated string-matching problem
  - finding all shifts s in the range $0 \leq s \leq n - m$ such that $P \sqsupset T_{s+m}$

Examples

- Assume $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
  - $\Sigma^* = \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{aa}, \mathtt{ab}, \mathtt{ac}, \mathtt{ba}, \mathtt{bb}, \mathtt{bc}, \mathtt{ca}, \mathtt{cb}, \mathtt{cc}, \ldots\}$
  - $x = \mathtt{ab}$ and $y = \mathtt{ba}$
  - $|x| = |y| = 2$
  - $xy = \mathtt{abba}$
  - $|xy| = |x| + |y| = 4$
  - $\epsilon \sqsubset \mathtt{abba}$ and $\epsilon \sqsupset \mathtt{abba}$
  - $\mathtt{a} \sqsubset \mathtt{abba}$ and $\mathtt{a} \sqsupset \mathtt{abba}$
  - $\mathtt{ab} \sqsubset \mathtt{abba}$ and $\mathtt{ba} \sqsupset \mathtt{abba}$
  - $\mathtt{abb} \sqsubset \mathtt{abba}$ and $\mathtt{bba} \sqsupset \mathtt{abba}$
  - $\mathtt{abba} \sqsubset \mathtt{abba}$ and $\mathtt{abba} \sqsupset \mathtt{abba}$

Overlapping-Suffix Lemma

- Assume x, y, and z are strings such that $x \sqsupset z$ and $y \sqsupset z$
  - if $|x| \leq |y|$, then $x \sqsupset y$
  - if $|x| \geq |y|$, then $y \sqsupset x$
  - if $|x| = |y|$, then $x = y$
- Proof



Naïve String-Matching Algorithm

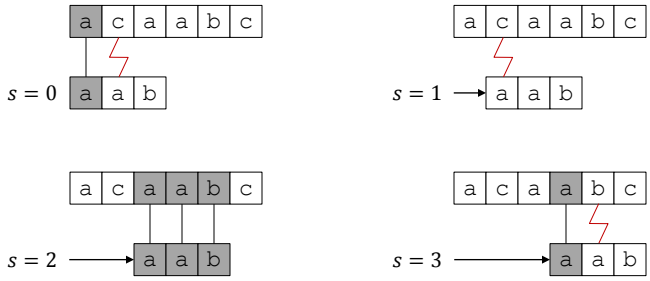
NAÏVE-STRING-MATCHER(T, P)

```
1  n ← length[T]
2  m ← length[P]
3  for s ← 0 to n − m
4       do if P[1..m] = T[s + 1..s + m]
5            then print "Pattern occurs with shift" s
```

- Comparing two stings (line 4) takes time $\Theta(t + 1)$
  - t denotes the number of matching characters
  - "$+1$" to cater for non-matching strings ($\neq O(0)$)
- Naïve algorithm takes time $O\big((n - m + 1)m\big)$
  - tight bound in the worst-case $\Theta\big((n - m + 1)m\big)$
  - consider matching text $\mathtt{a}^{\mathtt{n}}$ and the pattern $\mathtt{a}^{m}$
  - if $m = \lfloor n/2 \rfloor$, the worst-case running time is $\Theta(n^2)$

Example

- Graphical interpretation
  - sliding pattern over text in steps of length 1
  - noting for which shifts all of pattern characters equal the corresponding text characters



Rabin-Karp Algorithm

- Motivation
  - comparing numbers is “cheaper” than matching strings
  - represent text and pattern as numbers
  - use number-theoretic notions to match strings
- Assumptions and notation
  - $\Sigma_{10} = \{0,1,2, \ldots, 9\}$, but in the general case each character will be a digit in radix-d notation where $d = |\Sigma|$
  - p denotes the value corresponding to $P[1..m]$
  - given $T[1..n]$, t_s denotes the value of the length-m substring $T[s+1..s+m]$, for $s = 0,1, \ldots, n-m$
  - $t_s = p \Leftrightarrow T[s+1..s+m] = P[1..m]$

Rabin-Karp Algorithm

- Goal
  - compute p in time $\Theta(m)$
  - compute all t_s values in a total time $\Theta(n-m+1)$
  - get all valid shifts in time $\Theta(m)+\Theta(n-m+1)=\Theta(n)$
- Computing p from $P[1..m]$
  - can be done in $\Theta(m)$ using Horner's rule
  - $p=P[m]+d(P[m-1]+d(P[m-2]+\cdots+d(P[2]+dP[1])\cdots))$

Rabin-Karp Algorithm

- Computing t_0 from $T[1..n]$
  - use Horner's rule to compute t_0 in time $\Theta(m)$
- Computing $t_1, t_2, \ldots, t_{n-m}$ from $T[1..n]$
  - can be done in time $\Theta(n-m)$ since t_{s+1} can be computed from t_s in constant time
  - $t_{s+1}=d(t_s-d^{m-1}T[s+1])+T[s+m+1]$
- Example
  - assume Σ_{10}, $T=[3,1,4,1,5,9,2,6]$, $P=[1,4,1]$, and $m=3$
  - $p=141$, $t_0=314$
  - $t_1 = 10(t_0-10^2T[1])+T[4]$
    $= 10(314-300)+1=141$

All's Well That Ends Well

- Yes, Bill! But we're not done yet...
  - p and t_s may be too large to work with conveniently
  - assuming arithmetic operations on these numbers take "constant time" is unreasonable
- Simple solution
  - compute p and all t_s modulo a suitable modulus q
  - adding one operation does not change compute time
  - q is typically chosen as a prime such that dq fits within one computer word
  - $t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$, where $h \equiv d^{m-1} \pmod{q}$

Make It As Simple As Possible But Not Simpler

- Okay, Al! Maybe we went too far this time...
  - $t_s \equiv p \pmod{q}$ does not imply $t_s = p$
  - $t_s \not\equiv p \pmod{q}$ does imply $t_s \neq p$
- Example
  - Assume Σ_{10}, $p = 31415$, and $q = 13$
  - $31415 \equiv 7 \pmod{13}$
  - $67399 \equiv 7 \pmod{13}$
- Solution
  - use negative test as a fast heuristic to rule out invalid shifts
  - positive test must be validated to sort out **spurious hits**
  - if q is large, spurious hits are likely to occur less frequently

Rabin-Karp Algorithm

RABIN-KARP-MATCHER(T, P, d, q)

```
 1  n ← length[T]
 2  m ← length[P]
 3  h ← d^(m−1) mod q
 4  p ← 0
 5  t_0 ← 0
 6  for i ← 1 to m                                   Preprocessing
 7      do p ← (dp + P[i]) mod q
 8         t_0 ← (dt_0 + T[i]) mod q
 9  for s ← 0 to n − m                               Matching
10      do if p = t_s
11            then if P[1..m] = T[s + 1..s + m]
12                    then print "Pattern occurs with shift" s
13         if s < n − m
14            then t_{s+1} ← (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
```


Run-Time Analysis

- Worst case
  - $\Theta(m)$ to preprocess and $\Theta\big((n-m+1)m\big)$ to match
- Heuristic analysis of average case
  - "modulo q" acts as a random mapping from Σ^* to $\mathbb{Z}_q$
  - number of spurious hits expected to be $O(n/q)$ since the probability of $t_s \equiv p \pmod q$ can be estimated as $1/q$
  - expected matching time of Rabin-Karp algorithm

    no match: $O(n) + O\big(m(v + n/q)\big)$ match

    where v is the number of valid shifts
  - if $v = O(1)$ and $q \geq m$, the running time is $O(n+m)$ and since $m \leq n$ it is even expected to be $O(n)$!

String Matching with Finite Automata

- Idea
  - build a finite automaton to scan T for all occurrences of P
  - examine each character exactly once and in constant time
  - matching time $\Theta(n)$, but preprocessing time can be large
- A **finite automaton** M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$
  - Q is a finite set of **states**
  - $q_0 \in Q$ is the **start state**
  - $A \subseteq Q$ is a distinguished set of **accepting states**
  - Σ is a finite **input alphabet**
  - δ is a function from $Q \times \Sigma$ into Q, called **transition function** of M

String Matching with Finite Automata

- Finite automaton
  - begins in state q_0, reads one input character a at a time
  - **transitions** from state q into state $\delta(q, a)$
  - **accepts** the string read so far if current state $q \in A$
  - **reject** the string read so far if current state $q \notin A$
- A finite automaton induces a **final-state function** ϕ
  - $\phi: \Sigma^* \to Q$, such that $q = \phi(w)$ is the state M is in after scanning the string w
  - M accepts a string w if and only if $\phi(w) \in A$
  - recursive definition of ϕ

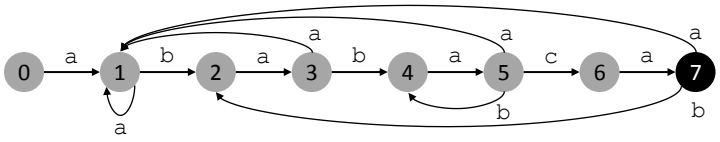
$$\phi(\epsilon) = q_0$$
$$\phi(wa) = \delta(\phi(w), a) \text{ for } w \in \Sigma^*, a \in \Sigma$$

String-Matching Automata

- For every pattern $P[1..m]$, we need to construct a string-matching automaton in preprocessing
  - the state set Q is $\{0,1,\ldots,m\}$, where start state q_0 is state 0 and state m is the only accepting state
  - the transition function is defined as $\delta(q,a) = \sigma(P_q a)$ for any state q and character a
- Suffix function σ for a given pattern $P[1..m]$
  - $\sigma: \Sigma \to \{0,1,\ldots,m\}$ such that $\sigma(x) = \max\{k: P_k \sqsupset x\}$ is the length of the longest prefix of P that is a suffix of x
  - for a pattern P of length m, $\sigma(x) = m$ if and only if $P \sqsupset x$
  - if $x \sqsupset y$, then $\sigma(x) \leq \sigma(y)$

Example

- Assume pattern $P =$ `ababaca`



  - 8 states and a "spine" of forward transitions
  - $\delta(1,\mathtt{a}) = 1$, since $P_1\mathtt{a} = \mathtt{a}\mathbf{a}$ and $\sigma(P_1\mathtt{a}) = 1$
  - $\delta(3,\mathtt{a}) = 1$, since $P_3\mathtt{a} = \mathtt{aba}\mathbf{a}$ and $\sigma(P_3\mathtt{a}) = 1$
  - $\delta(5,\mathtt{a}) = 1$ since $P_5\mathtt{a} = \mathtt{ababa}\mathbf{a}$ and $\sigma(P_5\mathtt{a}) = 1$
  - $\delta(5,\mathtt{b}) = 4$, since $P_5\mathtt{b} = \mathtt{ab}\mathbf{abab}$ and $\sigma(P_5\mathtt{b}) = 4$
  - $\delta(7,\mathtt{a}) = 1$, since $P_7\mathtt{a} = \mathtt{ababaca}\mathbf{a}$ and $\sigma(P_7\mathtt{a}) = 1$
  - $\delta(7,\mathtt{b}) = 2$, since $P_7\mathtt{b} = \mathtt{ababac}\mathbf{ab}$ and $\sigma(P_7\mathtt{b}) = 2$

String-Matching Automata

FINITE-AUTOMATON-MATCHER$(T, \mathrm{P}, \Sigma, m)$
1 $n \leftarrow length[T]$
2 $\delta \leftarrow$ COMPUTE-TRANSITION-FUNCTION(P, Σ)
3 $q \leftarrow 0$
4 **for** $i \leftarrow 1$ **to** n
5 **do** $q \leftarrow \delta(q, T[i])$
6 **if** $q = m$
7 **then** print "Pattern occurs with shift" $i - m$

- Matching time on a text of length n is $\Theta(n)$
  - simple loop structure with n iterations
  - does not account for the time required to compute the transition function δ

Computing the Transition Function δ

COMPUTE-TRANSITION-FUNCTION(P, Σ)
1 $m \leftarrow length[P]$
2 **for** $q \leftarrow 0$ **to** m
3 **do for** each character $a \in \Sigma$
4 **do** $k \leftarrow \min(m + 1, q + 2)$
5 **repeat** $k \leftarrow k - 1$
6 **until** $P_k \sqsupset P_q a$
7 $\delta(q, a) \leftarrow k$
8 **return** δ

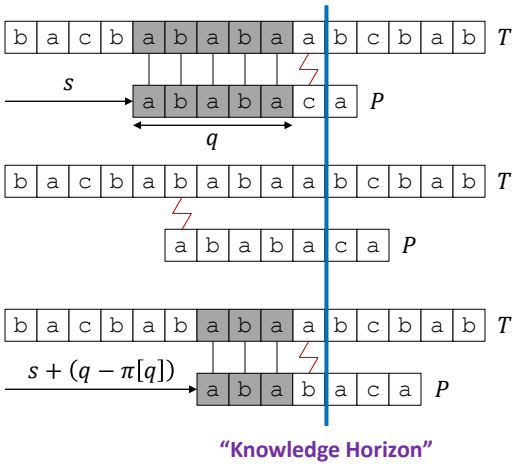
- Computing transition function takes time $O(m^3|\Sigma|)$
  - outer two **for** loops contribute a factor of $m^3|\Sigma|$
  - inner **repeat** loop can run at most $m + 1$ times
  - test $P_k \sqsupset P_q a$ can require up to m comparisons

Knuth-Morris-Pratt Algorithm

- Idea
  - avoid both computing transition function δ in time $O(m|\Sigma|)$ and testing useless shifts as in naïve algorithm
  - use auxiliary function $\pi[1..m]$ that can be pre-computed from the pattern in time $\Theta(m)$
  - array π allows δ to be computed efficiently "on the fly" as needed, in the *amortized* sense
- Prefix function π for a pattern $P[1..m]$
  - $\pi: \{1,2,\ldots,m\} \to \{0,1,\ldots,m-1\}$ such that $\pi[q] = \max\{k: k < q \text{ and } P_k \sqsupset P_q\}$
  - $\pi[q]$ is the length of the longest prefix of P that is a **proper** suffix of P_q

Example

- What's the next possible shift that should be tested?



i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

Knuth-Morris-Pratt Algorithm

```
KMP-MATCHER(T, P)
 1  n ← length[T]
 2  m ← length[P]
 3  π ← COMPUTE-PREFIX-FUNCTION(P)
 4  q ← 0                                  Number of characters matched
 5  for i ← 1 to n                         Scan the text from left to right
 6      do while q > 0 and P[q + 1] ≠ T[i]
 7            do q ← π[q]                  Next character does not match
 8         if P[q + 1] = T[i]
 9           then q ← q + 1                Next character matches
10         if q = m                        Is all of P matched?
11           then print "Pattern occurs with shift" i − m
12                q ← π[q]                 Look for the next match
```


Computing the Prefix Function π

```
COMPUTE-PREFIX-FUNCTION(P)
 1  m ← length[P]
 2  π[1] ← 0
 3  k ← 0
 4  for q ← 2 to m
 5      do while k > 0 and P[k + 1] ≠ P[q]
 6            do k ← π[k]
 7         if P[k + 1] = P[q]
 8           then k ← k + 1
 9         π[q] ← k
10  return π
```

Run-Time Analysis

- Computing the prefix function takes time $\Theta(m)$
  - outer **for** loop takes time $\Theta(m)$
  - amortized cost of **for** loop body is $O(1)$
    - *amortized analysis* with a potential of k, corresponding to the current state of k in the algorithm
    - in each iteration of the **for** loop, k increases at most by 1
    - since $\pi[k] < k$, there is a decrease of k for each increase of k
- String-matching takes time $\Theta(n)$
  - with q as the potential function, the same amortized argument as above can be made for the matching time