Chapter 32: String Matching

Fall 2007

Simonas Šaltenis
simas@cs.aau.dk

Modified by Pierre Flener (version of 30 November 2016)

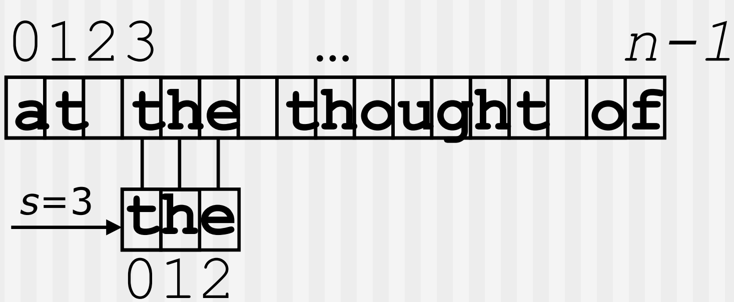
String Matching Algorithms

- Goals of the lecture:
  - Naïve string matching algorithm and analysis
  - **Rabin-Karp** algorithm (1987) and its analysis
  - **Knuth-Morris-Pratt** algorithm (1977) ideas

- Turing Awards:
  - 1974: Donald Knuth
  - 1976: Michael Rabin
  - 1985: Richard Karp

String Matching Problem

- Input:
  - *Text* T = "**`at the thought of`**"
    - n = length(T) = 17
  - *Pattern* P = "**`the`**"
    - m = length(P) = 3   We assume $m \le n$.
- Output: (CLRS indexes from 1 & aims at all shifts)
  - *Shift* s – the smallest integer ($0 \le s \le n-m$) such that $T[s\ ..\ s+m-1] = P[0\ ..\ m-1]$. Returns –1 if no such s exists.



Naïve String Matching

- Idea: Brute force
  - Check all values of *s* from 0 to *n*–*m*

```
Naïve-Matcher(T,P)
for s ← 0 to n – m do
   j ← 0
   // check if T[s..s+m–1] = P[0..m–1]
   while T[s+j] = P[j] do
      j ← j + 1
      if j = m then return s
return –1
```

- Let *T* = "**at the thought of**" and *P* = "**though**"
  - *What is the number of character comparisons*?

Analysis of Naïve String Matching

- The analysis is made for finding all shifts
- Worst case:
  - Outer loop: $n-m+1$ iterations
  - Inner loop: max m constant-time iterations
  - Total: max $(n-m+1)m = O(nm)$, as $m \leq n$
  - What input gives this worst-case behaviour?
- Best case: $\Theta(n-m+1)$
  - When?
- Completely random text and pattern:
  - $O(n-m)$

Analysis of Naïve String Matching

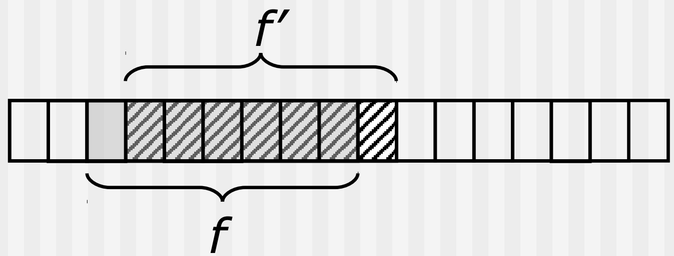
- The analysis is made for finding all shifts
- Worst case:
  - Outer loop: $n-m+1$ iterations
  - Inner loop: max m constant-time iterations
  - Total: max $(n-m+1)m = O(nm)$, as $m \leq n$
  - What input gives this worst-case behaviour?
    Examples: $P=a^m$ and $T=a^n$; $P=a^{m-1}b$ and $T=a^n$
- Best case: $\Theta(n-m+1)$
  - When?
- Completely random text and pattern:
  - $O(n-m)$

Analysis of Naïve String Matching

- The analysis is made for finding all shifts
- Worst case:
  - Outer loop: $n-m+1$ iterations
  - Inner loop: max m constant-time iterations
  - Total: max $(n-m+1)m = O(nm)$, as $m \le n$
  - What input gives this worst-case behaviour? Examples: $P=a^m$ and $T=a^n$; $P=a^{m-1}b$ and $T=a^n$
- Best case: $\Theta(n-m+1)$
  - When? Example: $P[0]$ is not in T
- Completely random text and pattern:
  - $O(n-m)$

Fingerprint Idea

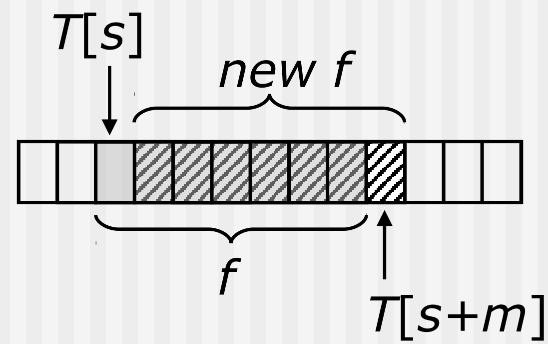
- Assume:
  - We can compute a ***fingerprint*** $f(P)$ of P in $\Theta(m)$ time; similarly for $f(T[0 .. m-1])$
  - $f(P) \neq f(t) \Rightarrow P \neq t$ for any $t = T[s .. s+m-1]$ (*)
  - We can compare fingerprints in $O(1)$ time
  - We can compute $f' = f(T[s+1 .. s+m])$ from $f(T[s .. s+m-1])$ in $O(1)$ time



Algorithm with Fingerprints

- Let the alphabet $\Sigma=\{$**0,1,2,3,4,5,6,7,8,9**$\}$
- Let the fingerprint be a decimal number, i.e., *f*("**2045**") = $2*10^3 + 0*10^2 + 4*10^1 + 5 = 2045$

```
Fingerprint-Matcher(T,P)
fp ← compute f(P)
ft ← compute f(T[0..m-1])
for s ← 0 to n – m do
   if fp = ft then return s
   ft ← (ft - T[s]*10^(m-1))*10 + T[s+m]
return -1
```



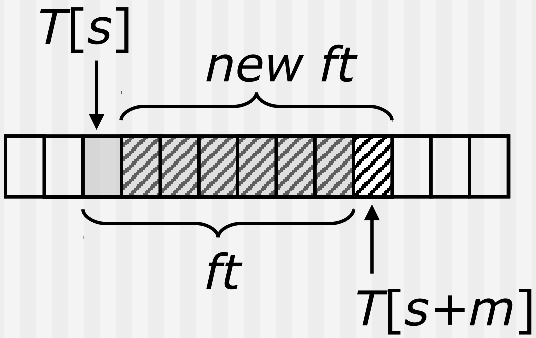
- Running time: $2\Theta(m) + \Theta(n-m) = \Theta(n)$, as $m \leq n$
- *Where is the catch*?! There are *two*, actually.

Using a Hash Function

- First problem: We cannot assume m-digit number arithmetic works in $O(1)$ time!
- Solution = hashing: $h(s) = f(s) \bmod q$
  - Example: if $q=7$, then $h(\text{"52"}) = 52 \bmod 7 = 3$
  - We now indeed have: $h(P) \neq h(t) \Rightarrow P \neq t$
- Second problem: the inverse contrapositive "$f(P)=f(t) \Rightarrow P=t$" of (*) was not assumed!
  - Example: if $q=7$ then $h(\text{"}\mathbf{59}\text{"})=3$, but "**59**"≠"**52**"
- Basic "mod q" arithmetic:
  - $(a+b) \bmod q = (a \bmod q + b \bmod q) \bmod q$
  - $(a*b) \bmod q = (a \bmod q) * (b \bmod q) \bmod q$

Preprocessing and Stepping

- Preprocessing, using Horner's rule and 'mod' laws:
  - fp = $(10*(\ldots*(10*(10*0+P[0])+P[1])+\ldots)+P[m-1])\bmod q$
  - In the same way, compute ft from $T[0..m-1]$
  - *Exercise*: Let P = "**2531**" and q = 7: what is fp?
- Stepping:
  - $ft \leftarrow (ft - T[s]*10^{m-1} \bmod q)*10 + T[s+m]) \bmod q$
  - $10^{m-1} \bmod q$ can be computed *once*, in the preprocessing
  - *Exercise*: Let $T[\ldots]$ = "**5319**" and q = 7: what is the new ft when $T[s+m]$="**7**"?



Rabin-Karp Algorithm (1987)

```
Rabin-Karp-Matcher(T,P)
q ← a prime larger than m
c ← 10^(m-1) mod q  // run a loop multiplying by 10 mod q
fp ← 0; ft ← 0
for i ← 0 to m-1 do  // preprocessing
   fp ← (10*fp + P[i]) mod q
   ft ← (10*ft + T[i]) mod q
for s ← 0 to n – m do  // matching
   if fp = ft then   // run a loop to compare strings
      if P[0..m-1] = T[s..s+m-1] then return s
   ft ← ((ft – T[s]*c)*10 + T[s+m]) mod q
return –1
```

- *Exercise: How many character comparisons are done if T =* “**2531978**”, *P =* “**1978**”, and *q* = 7?

Analysis

- If q is a prime number, then the hash function distributes m-digit strings *evenly* among the q values.
  - Thus, only every q^{th} value of shift s will result in matching fingerprints, which requires comparing strings with $O(m)$ comparisons
- Expected running time, if $q > m$:
  - Preprocessing: $\Theta(m)$
  - Outer loop: $n-m+1$ iterations
  - All inner loops: maximum $\frac{n-m}{q} m = O(n-m)$
  - Total time: $O(n+m) = O(n)$
- Worst-case running time: $O(nm)$

Rabin-Karp in Practice

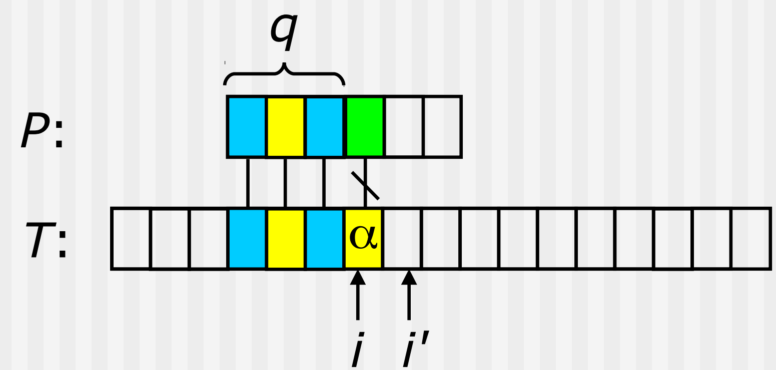
- If the alphabet has d characters, then interpret characters as radix-d digits: replace 10 by d in the algorithm.
- Choosing a prime number $q > m$ can be done with a randomised algorithm in $O(m)$ time, or q can be fixed to be the largest prime so that $d*q$ fits in a computer word.
- Rabin-Karp is simple and can be extended to two-dimensional pattern matching.

Matching in *n* Comparisons

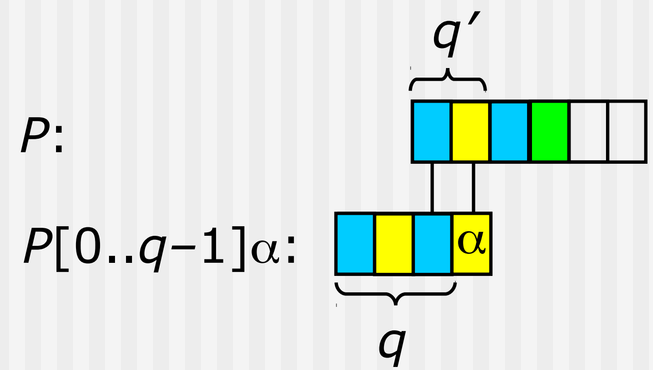
- Goal: Each text character is compared only once to a pattern character.
- Problem with the naïve algorithm:
  - Forgets what was learned from a partial match!
  - Examples:
    - *T* = "**`Tweedledee and Tweedledum`**" and *P* = "**`Tweedledum`**"
    - *T* = "***`pappappappar`***" and *P* = "**`pappar`**"

General Situation

- State of the algorithm:
  - Reading character $T[i]$
  - $q<m$ characters of P are matched so far in T
  - We see a non-matching character α in $T[i]$



- Need to find for $i'=i+1$:
  - Length of longest prefix of P that is a suffix of $P[0..q-1]\alpha$:



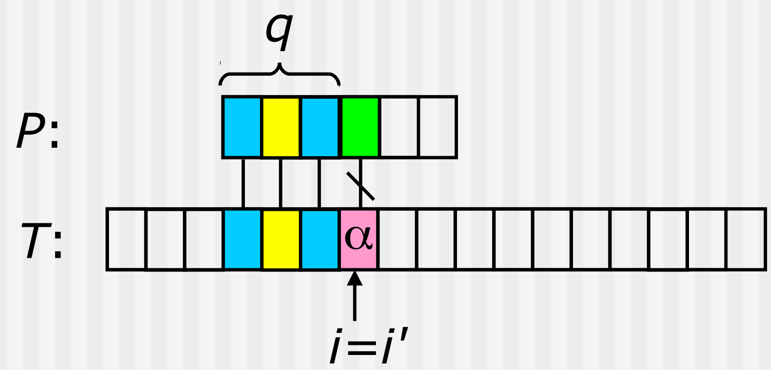
    new $q = q' = \max\{k \leq q \mid P[0..k-1] = P[q-k+1..q-1]\alpha\}$
  - Pre-computation would take $O(m|\Sigma|)$ time and memory...

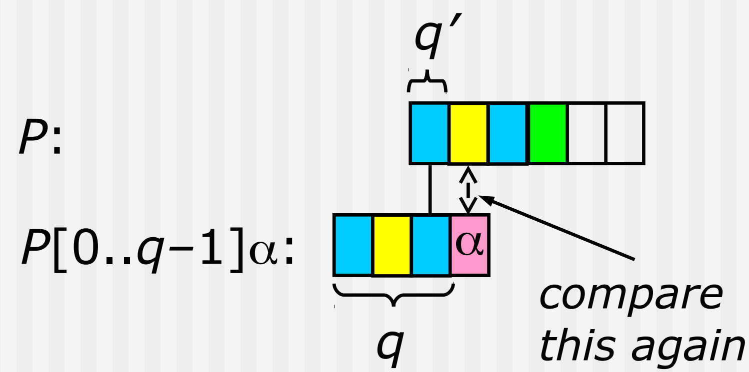
Finite Automaton Search

- Algorithm:
  - Preprocess:
    - For each q ($0 \leq q \leq m-1$) and each $\alpha \in \Sigma$ pre-compute a new value of q. Let us call it $\sigma(q,\alpha)$.
    - Fill a table of size $m|\Sigma|$
  - Run through the text
    - Whenever a mismatch is found ($P[q] \neq T[s+q]$):
    - Set $s = s + q - \sigma(q,\alpha) + 1$ and $q = \sigma(q,\alpha)$
- Analysis:
  - ☺ Matching phase in $O(n)$ time
  - ☹ Too much memory: $\Theta(m|\Sigma|)$,
    too much preprocessing: at best $O(m|\Sigma|)$.

Prefix Function

- Idea: Revisit the unmatched character (α)!
- State of the algorithm:
  - Reading character $T[i]$
  - $q<m$ characters of P are matched
  - We see a non-matching character α in $T[i]$
- Need to find for $i' = i$:
  - Length of the longest prefix of $P[0..q-2]$ that is a suffix of $P[0..q-1]$:





$$\text{new } q = q' = \pi\,[q] = \max\{k < q \mid P[0..k-1] = P[q-k..q-1]\}$$

Prefix Table

- Pre-compute a *prefix table* of size m to store the values of $\pi[q]$ for $0 \leq q \leq m$

P		**p**	**a**	**p**	**p**	**a**	**r**
q	0	1	2	3	4	5	6
$\pi[q]$	0	0	0	1	1	2	0

- *Exercise:*
  Compute a prefix table for P = "**dadadu**"

Knuth-Morris-Pratt (1977)

```
KMP-Matcher(T,P)
01 π ← Compute-Prefix-Table(P)
02 q ← 0 // number of chars matched = index of next char
03 for i ← 0 to n-1 do // scan text from left to right
04     while q > 0 and P[q] ≠ T[i] do
05         q ← π[q]
06     if P[q] = T[i] then q ← q+1
07     if q = m then return i−m+1
08 return −1
```

To return all shifts, replace the **`then`** block of line 07 by

```
    print i−m+1; q ← π[q]
```

`Compute-Prefix-Table` is essentially the KMP matching algorithm, but performed on *P* as text.

Analysis of KMP

- Worst-case running time: $O(n+m) = O(n)$
  - Main algorithm: $O(n)$
  - **`Compute-Prefix-Table`**: $O(m)$
- Space usage: $O(m)$

Reverse Naïve Algorithm

- Why not search from the end of *P*?
  - Boyer and Moore

```
Reverse-Naïve-Matcher(T,P)
for s ← 0 to n-m
   j ← m-1   // start from the end
   // check if T[s..s+m-1] = P[0..m-1]
   while T[s+j] = P[j] do
      j ← j-1
      if j < 0 return s
return -1
```

- Running time is exactly the same as for the naïve algorithm…

Occurrence Heuristic

- Boyer and Moore added two heuristics to the reverse naïve matcher, to get an $O(n+m)$ algorithm, but it is complex
- Horspool suggested just to use the modified *occurrence* heuristic:
  - *After a mismatch, align* $T[s + m-1]$ *with the rightmost occurrence of that letter in the pattern* $P[0..m-2]$
  - Examples:
    - T= "**`detective date`**" and P= "**`date`**"
    - T= "**`tea kettle`**" and P= "**`kettle`**"

Shift Table

- In preprocessing, compute the shift table of the size $|\Sigma|$.

$$\text{shift}[w] = \begin{cases} m - 1 - \max\{i < m-1 \mid P[i] = w\} & \text{if } w \text{ is in } P[0..m-2], \\ m & \text{otherwise.} \end{cases}$$

- *Example*: *P* = "**`kettle`**"
  - shift[**`e`**] =4, shift[**`l`**] =1, shift[**`t`**] =2, shift[**`k`**] =5
  - shift[any other letter] = 6
- *Exercise*: *P* = "**`pappar`**"
  - What is the shift table?

Boyer-Moore-Horspool

```
BMH-Matcher(T,P)
// compute the shift table for P
for c ← 0 to |Σ|- 1 do
   shift[c] = m          // default values
for k ← 0 to m-2 do
   shift[P[k]] = m-1-k
// search
s ← 0
while s ≤ n-m do
   j ← m-1   // start from the end
   // check if T[s..s+m-1] = P[0..m-1]
   while T[s+j] = P[j] do
      j ← j 1
      if j < 0 then return s
   s ← s + shift[T[s+m-1]]  // shift by last letter
return -1
```

BMH Analysis

- Worst-case running time
  - Preprocessing: $O(|\Sigma|+m)$
  - Searching: $O(nm)$
    - *Exercise: What input gives this bound*?
  - Total: $O(nm)$
- Space: $O(|\Sigma|)$
  - Independent of m
- On real-world data sets: very fast

Comparison

- Let us compare the algorithms.
  Criteria:
  - Worst-case running time
    - Preprocessing
    - Searching
  - Expected running time
  - Space used
  - Implementation complexity