Advanced Algorithm Design and Analysis (Lecture 3)

SW5 fall 2004

Simonas Šaltenis

E1-215b

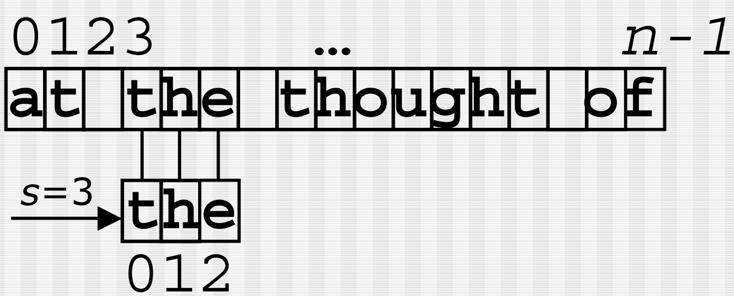
simas@cs.aau.dk

Text-search Algorithms

- Goals of the lecture:
  - *Naive text-search algorithm and its analysis;*
  - ***Rabin-Karp** algorithm and its analysis;*
  - ***Knuth-Morris-Pratt** algorithm ideas;*
  - ***Boyer-Moore-Horspool** algorithm and its analysis.*
  - *Comparison of the **advantages and disadvantages** of the different text-search algorithms.*

Text-Search Problem

- Input:
  - *Text* *T* = "**`at the thought of`**"
    - *n* = length(*T*) = 17
  - *Pattern* *P* = "**`the`**"
    - *m* = length(*P*) = 3
- Output:
  - *Shift* *s* – the smallest integer ($0 \leq s \leq n - m$) such that $T[s \,..\, s+m-1] = P[0 \,..\, m-1]$. Returns –1, if no such *s* exists



Naïve Text Search

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

```
Naive-Search(T,P)
for s ← 0 to n - m
   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 return s
return -1
```

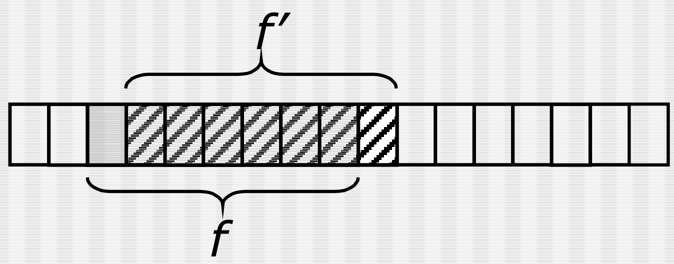
- Let *T* = "**`at the thought of`**" and *P* = "**`though`**"
  - What is the number of character comparisons?

Analysis of Naïve Text Search

- Worst-case:
  - Outer loop: $n - m$
  - Inner loop: m
  - Total $(n-m)m = O(nm)$
  - What is the input the gives this worst-case behaviuor?
- Best-case: n-m
  - When?
- Completely random text and pattern:
  - $O(n-m)$

Fingerprint idea

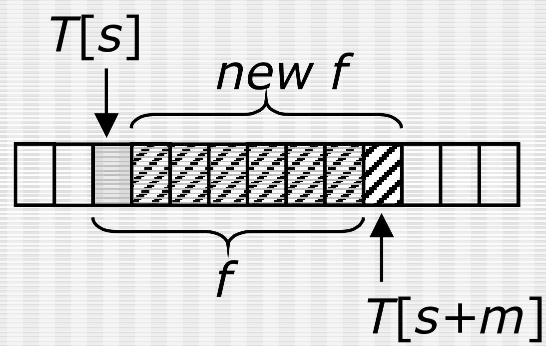
- Assume:
  - We can compute a ***fingerprint*** $f(P)$ of P in $O(m)$ time.
  - If $f(P) \neq f(T[s .. s+m-1])$, then $P \neq T[s .. s+m-1]$
  - We can compare fingerprints in O(1)
  - We can compute $f' = f(T[s+1.. s+m])$ from $f(T[s .. s+m-1])$, in O(1)



Algorithm with Fingerprints

- Let the alphabet $\Sigma=\{0,1,2,3,4,5,6,7,8,9\}$
- Let fingerprint to be just a decimal number, i.e., $f(\text{“1045”}) = 1*10^3 + 0*10^2 + 4*10^1 + 5 = 1045$

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



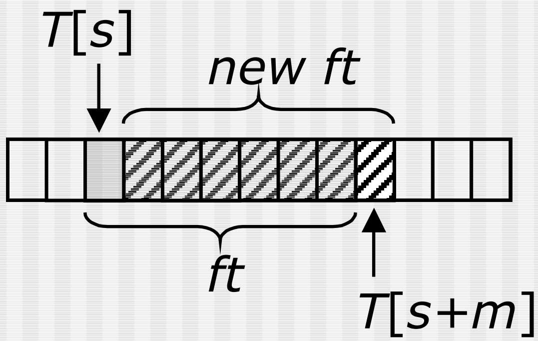
- Running time $2O(m) + O(n-m) = O(n)$!
- Where is the catch?

Using a Hash Function

- Problem:
  - we can not assume we can do arithmetics with m-digits-long numbers in O(1) time
- Solution: Use a hash function $h = f \bmod q$
  - For example, if $q = 7$, h("**52**") = 52 mod 7 = 3
  - $h(S_1) \neq h(S_2) \Rightarrow S_1 \neq S_2$
  - But $h(S_1) = h(S_2)$ *does not* imply $S_1 = S_2$!
    - For example, if $q = 7$, h("**73**") = 3, but "**73**" ≠ "**52**"
- Basic "mod q" arithmetics:
  - $(a+b) \bmod q = (a \bmod q + b \bmod q) \bmod q$
  - $(a*b) \bmod q = (a \bmod q)*(b \bmod q) \bmod q$

Preprocessing and Stepping

- Preprocessing:
  - *fp* = $P[m\text{-}1] + 10*(P[m\text{-}2] + 10*(P[m\text{-}3]+ \ldots$
    $\ldots + 10*(P[1] + 10*P[0])\ldots))$ mod q
  - In the same way compute *ft* from $T[0..m\text{-}1]$
  - Example: P = "**2531**", $q = 7$, what is *fp*?
- Stepping:
  - $ft = (ft - T[s]*10^{m-1} \bmod q)*10 + T[s+m]) \bmod q$
  - $10^{m-1} \bmod q$ can be computed once in the preprocessing
  - Example: Let $T[\ldots]$ = "**5319**", $q = 7$, what is the corresponding *ft*?



Rabin-Karp Algorithm

```
Rabin-Karp-Search(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  // preprocessing
   fp ← (10*fp + P[i]) mod q
   ft ← (10*ft + T[i]) mod q
for s ← 0 to n – m  // matching
   if fp = ft then   // run a loop to compare strings
      if P[0..m-1] = T[s..s+m-1] return s
   ft ←((ft – T[s]*c)*10 + T[s+m]) mod q
return –1
```

- How many character comparisons are done if *T* = "`2531978`" and *P* = "`1978`"?

Analysis

- If q is a prime, 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 will require comparing stings with $O(m)$ comparisons)
- Expected running time (if $q > m$):
  - Preprocessing: $O(m)$
  - Outer loop: $O(n-m)$
  - All inner loops: $\frac{n-m}{q}m = O(n-m)$
  - Total time: $O(n-m)$
- Worst-case running time: $O(nm)$

Rabin-Karp in Practice

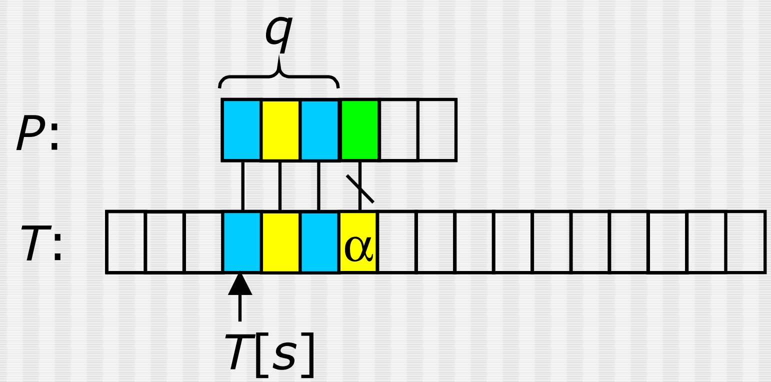
- If the alphabet has d characters, interpret characters as radix-d digits (replace 10 with d in the algorithm).
- Choosing prime $q > m$ can be done with randomized algorithms in O(m), or q can be fixed to be the largest prime so that $10*q$ fits in a computer word.
- Rabin-Karp is simple and can be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

Searching in *n* comparisons

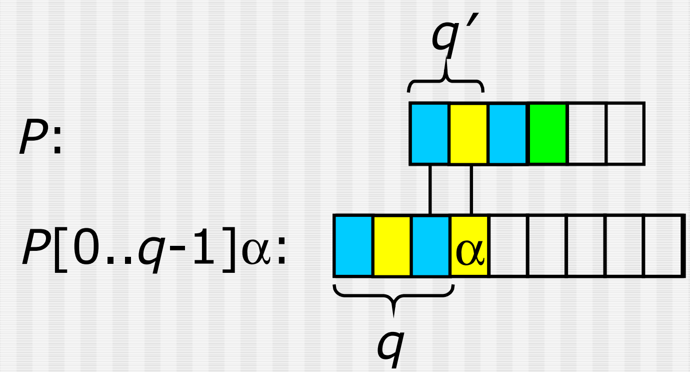
- The goal: each character of the text is compared only once!
- Problem with the naïve algorithm:
  - Forgets what was learned from a partial match!
  - Examples:
    - *T* = "**`Tweedledee and Tweedledum`**" and *P* = "**`Tweedledum`**"
    - *T* = "**`pappar`**" and *P* = "**`pappappappar`**"

General situation

- State of the algorithm:
  - Checking shift s,
  - q characters of P are matched,
  - we see a non-matching character α in T.



- Need to find:
  - Largest prefix "P-" such that it is a suffix of $P[0..q\text{-}1]\alpha$:
    - New $q' = \max\{k \le q \mid P[0..k-1] = P[q-k+1..q-1]\alpha\}$

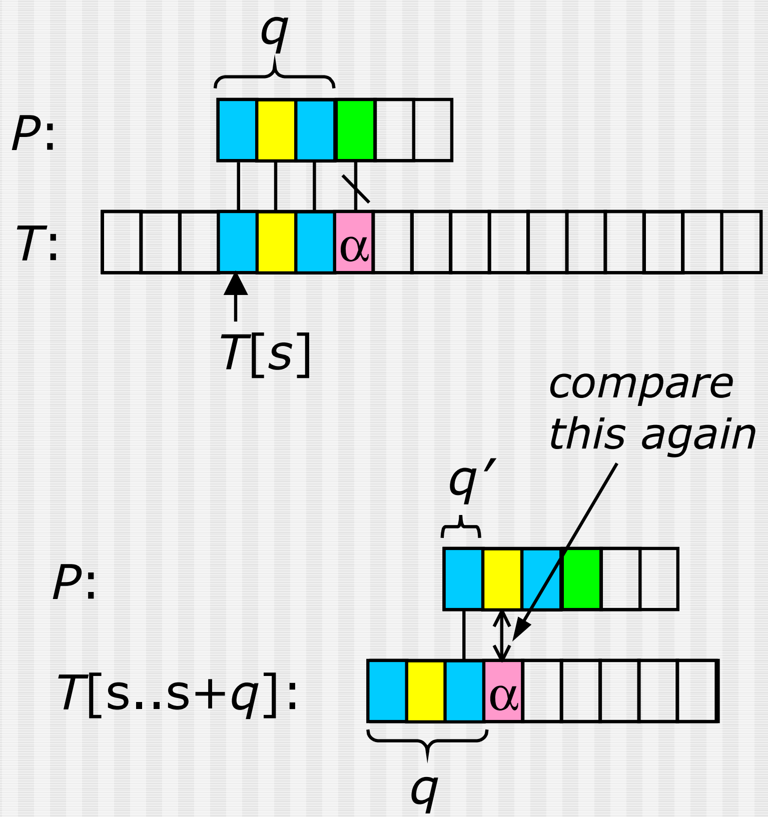


Finite automaton search

- Algorithm:
  - Preprocess:
    - For each q ($0 \leq q \leq m-1$) and each $\alpha \in \Sigma$ pre-compute a new value of q. Let's call it $\sigma(q,\alpha)$
    - Fills a table of a 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 $O(n)$
  - ☹ Too much memory: $O(m|\Sigma|)$, two much preprocessing: at best $O(m|\Sigma|)$.

Prefix function

- Idea: forget unmatched character (α)!
- State of the algorithm:
  - Checking shift *s*,
  - *q* characters of *P* are matched,
  - we see a non-matching character α in *T*.
- Need to find:
  - Largest prefix "*P*-" such that it is a suffix of *P*[0..*q*-1]:
    - New $q' = \pi[q] = \max\{k < q \mid P[0..k-1] = P[q-k..q-1]\}$



Prefix table

- We can pre-compute a *prefix table* of size m to store values of $\pi[q]$ $(0 \leq q < m)$

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

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

Knuth-Morris-Pratt Algorithm

```
KMP-Search(T,P)
π ← Compute-Prefix-Table(P)
q ← 0       // number of characters matched
for i ← 0 to n-1  // scan the text from left to right
   while q > 0 and P[q] ≠ T[i] do
      q ← π[q]
   if P[q] = T[i] then q ← q + 1
   if q = m then return i – m + 1
return –1
```

- **`Compute-Prefix-Table`** is the essentially the same KMP search algorithm performed on P.

Analysis of KMP

- Worst-case running time: $O(n+m)$
  - 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-Naive-Search(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 of the naïve algorithm…

Occurrence heuristic

- Boyer and Moore added two heuristics to reverse naïve, to get an $O(n+m)$ algorithm, but its 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[`t`] =5
  - shift[any other letter] = 6
- Example: *P* = "**`pappar`**"
  - What is the shift table?

Boyer-Moore-Horspool Alg.

```
BMH-Search(T,P)
// compute the shift table for P
for c ← 0 to |Σ|- 1
   shift[c] = m        // default values
for k ← 0 to m - 2
   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 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: $O(nm)$
    - What input gives this bound?
  - Total: $O(nm)$
- Space: $O(|\Sigma|)$
  - Independent of m
- On real-world data sets very fast

Comparison

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