

دانشگاه آزاد اسلامی واحد تبریز



نام درس: طراحی و تحلیل الگوریتم های پیشرفته

بخش: جستجوی الگودرشته

نام استاد: دکتر مسعود کارگر

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"

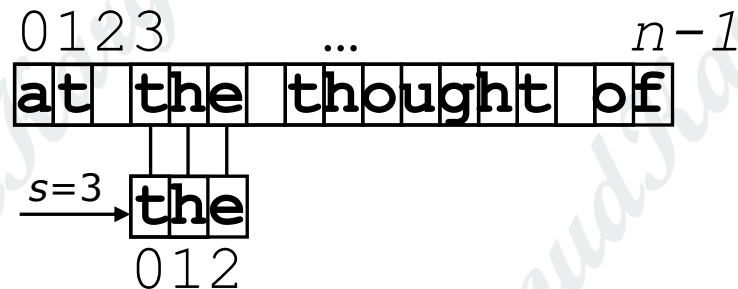
- $n = \text{length}(T) = 17$

- Pattern P = "the"

- $m = \text{length}(P) = 3$

■ 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)

```
01 for  $s \leftarrow 0$  to  $n - m$ 
02      $j \leftarrow 0$ 
03     // check if  $T[s..s+m-1] = P[0..m-1]$ 
04     while  $T[s+j] = P[j]$  do
05          $j \leftarrow j + 1$ 
06         if  $j = m$  return  $s$ 
07 return -1
```

- Let $T = \text{"at the thought of"}$ and $P = \text{"though"}$
 - What is the number of character comparisons?

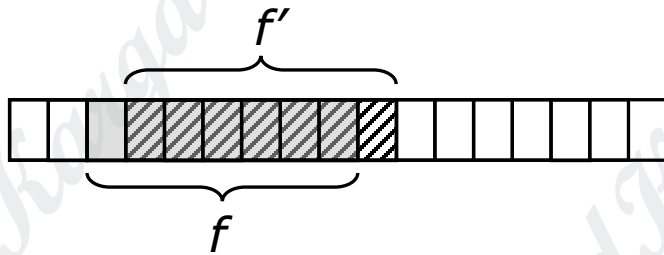
Analysis of Naïve Text Search

- Worst-case:
 - Outer loop: $n - m$
 - Inner loop: m
 - Total $(n-m)m = O(nm)$
 - What is the input that gives this worst-case behavior?
- 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 $O(1)$

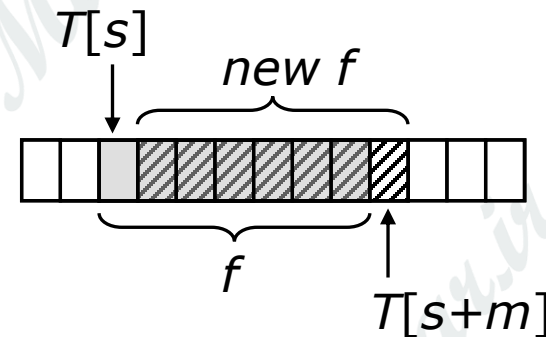


Algorithm with Fingerprints

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

Fingerprint-Search(T, P)

```
01 fp ← compute f(P)
02 f ← compute f(T[0..m-1])
03 for s ← 0 to n - m do
04     if fp = f return s
05     f ← (f - T[s] * 10m-1) * 10 + T[s+m]
06 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 $q = 7$, $h("52") = 52 \bmod 7 = 3$
- $h(S_1) \neq h(S_2) \Rightarrow S_1 \neq S_2$
- But $h(S_1) = h(S_2)$ *does not* imply $S_1 = S_2$!
 - For example, if $q = 7$, $h("73") = 3$, but $"73" \neq "52"$

■ Basic "mod q " arithmetics:

- $(a+b) \bmod q = (a \bmod q + b \bmod q) \bmod q$
- $(a*b) \bmod q = (a \bmod q)*(b \bmod q) \bmod q$

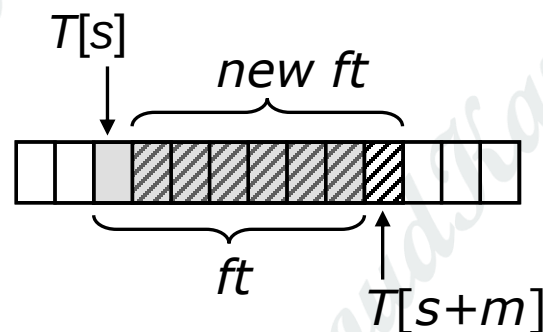
Preprocessing and Stepping

■ Preprocessing:

- $fp = P[m-1] + 10*(P[m-2] + 10*(P[m-3] + \dots + 10*(P[1] + 10*P[0])\dots)) \bmod q$
- In the same way compute ft from $T[0..m-1]$
- Example: $P = \text{"2531"} , q = 7$, 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 $T[\dots] = \text{"5319"} , q = 7$, what is the corresponding ft ?



Rabin-Karp Algorithm

Rabin-Karp-Search (T, P)

```
01 q ← a prime larger than m
02 c ← 10m-1 mod q // run a loop multiplying by 10 mod q
03 fp ← 0; ft ← 0
04 for i ← 0 to m-1 // preprocessing
05     fp ← (10*fp + P[i]) mod q
06     ft ← (10*ft + T[i]) mod q
07 for s ← 0 to n - m // matching
08     if fp = ft then // run a loop to compare strings
09         if P[0..m-1] = T[s..s+m-1] return s
10     ft ← ((ft - T[s]*c)*10 + T[s+m]) mod q
11 return -1
```

- How many character comparisons are done if $T = \text{"2531978"}$ and $P = \text{"1978"}$?

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 strings with $O(m)$ comparisons)
- Expected running time (if $q > m$):
 - Preprocessing: $O(m)$
 - Outer loop: $O(n-m)$
 - All inner loops:
 - 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 $10 \cdot q$ fits in a computer word.
- Rabin-Karp is simple and can be easily extended to two-dimensional pattern matching.

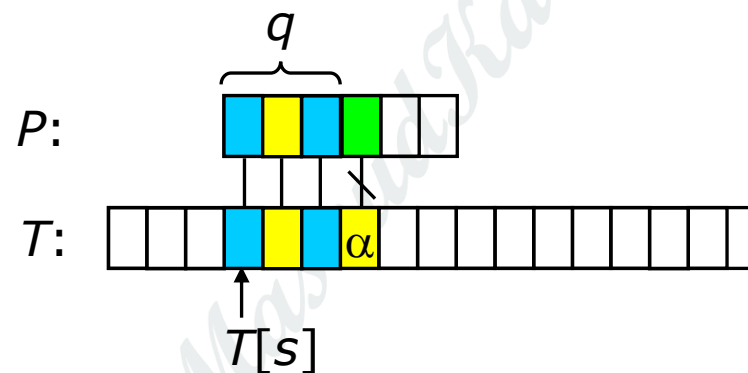
Searching in n comparisons

- 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:
 - $T = \text{"Tweedledee and Tweedledum"}$ and $P = \text{"Tweedledum"}$
 - $T = \text{"pappar"}$ and $P = \text{"pappappappar"}$

General situation

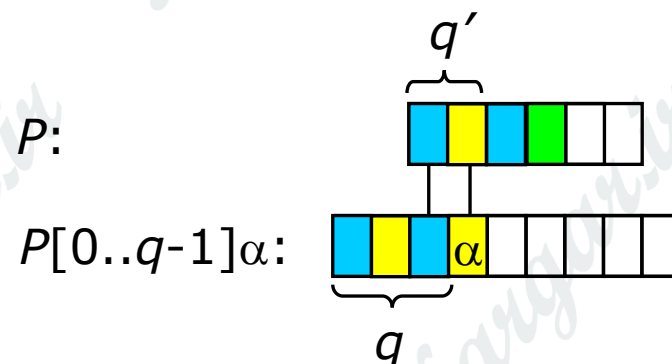
■ State of the algorithm:

- 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 $P[0..q-1]\alpha$:



- New $q' = \max\{k \leq q \mid P[0..k-1] = P[q-k+1..q-1]\alpha\}$

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: $O(m|\Sigma|)$, too much preprocessing: at best $O(m|\Sigma|)$.

Prefix function

- Idea: forget unmatched character (α)!

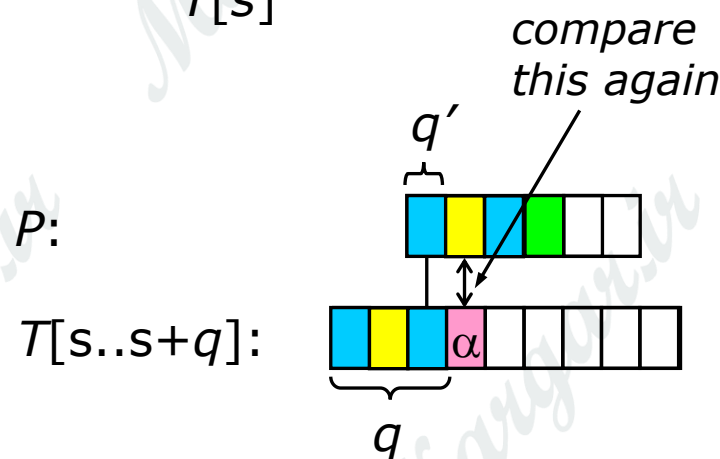
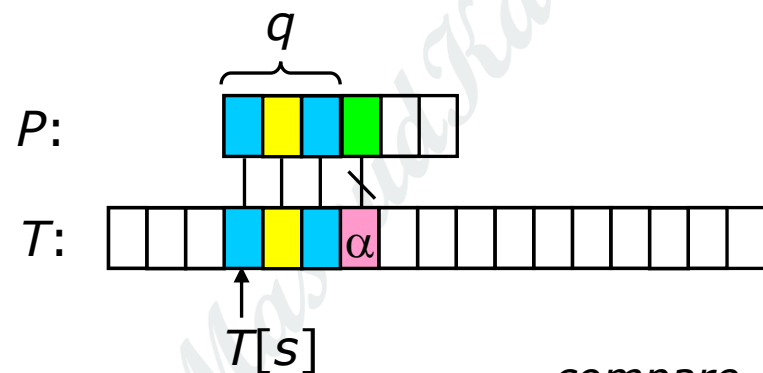
- State of the algorithm:

- 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: $P = \text{"dadadu"}$

Knuth-Morris-Pratt Algorithm

KMP-Search (T, P)

```
01  $\pi \leftarrow \text{Compute-Prefix-Table}(P)$ 
02  $q \leftarrow 0$  // number of characters matched
03 for  $i \leftarrow 0$  to  $n-1$  // scan the text from left to right
04     while  $q > 0$  and  $P[q] \neq T[i]$  do
05          $q \leftarrow \pi[q]$ 
06     if  $P[q] = T[i]$  then  $q \leftarrow q + 1$ 
07     if  $q = m$  then return  $i - m + 1$ 
08 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)

```
01 for  $s \leftarrow 0$  to  $n - m$ 
02      $j \leftarrow m - 1$  // start from the end
03     // check if  $T[s..s+m-1] = P[0..m-1]$ 
04     while  $T[s+j] = P[j]$  do
05          $j \leftarrow j - 1$ 
06         if  $j < 0$  return  $s$ 
07 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:
 - $T = \text{"detective date"}$ and $P = \text{"date"}$
 - $T = \text{"tea kettle"}$ and $P = \text{"kettle"}$

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: $P = \text{"kettle"}$
 - $\text{shift}[\text{e}] = 4, \text{shift}[\text{l}] = 1, \text{shift}[\text{t}] = 2, \text{shift}[\text{t}] = 5$
 - $\text{shift}[\text{any other letter}] = 6$
- Example: $P = \text{"pappar"}$
 - What is the shift table?

Boyer-Moore-Horspool Alg.

BMH-Search (T, P)

```
01 // compute the shift table for P
01 for c  $\leftarrow$  0 to  $|\Sigma| - 1$ 
02     shift[c] = m // default values
03 for k  $\leftarrow$  0 to m - 2
04     shift[P[k]] = m - 1 - k
05 // search
06 s  $\leftarrow$  0
07 while s  $\leq$  n - m do
08     j  $\leftarrow$  m - 1 // start from the end
09     // check if T[s..s+m-1] = P[0..m-1]
10     while T[s+j] = P[j] do
11         j  $\leftarrow$  j - 1
12     if j < 0 return s
13     s  $\leftarrow$  s + shift[T[s + m-1]] // shift by last letter
14 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

قدردانی

Simonas Šaltenis