# 算法设计与分析—入门篇

## 第七讲 字符串搜索

哈尔滨工业大学
王宏志
wangzh@hit.edu.cn
http://homepage.hit.edu.cn/pages/wang/

# 提纲

7.1 字符串搜索概述

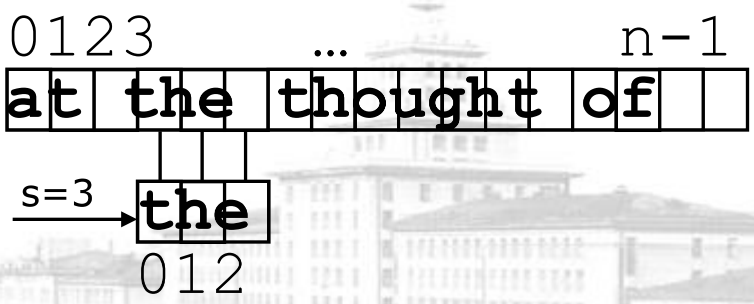
7.2 Rabin-Karp算法

7.3 KMP算法

7.4 BMH算法

# 字符串匹配问题

- 输入:
  - *文本* $T$ = “**at the thought of**”
    - $n$ = length($T$) = 17
  - *模式* $P$ = “**the**”
    - $m$ = length($P$) = 3
- 输出:
  - *移动到* $s$ − 最小的整数 ($0 \le s \le n - m$) 满足 $T[s .. s+m-1] = P[0 .. m-1]$. 返回 −1, 如果不存在这样的 $s$



# 简单匹配算法

- 想法：暴力搜索
  - 检查从0 到 n – m的所有值

```
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
```

- 令 T = “at the thought of”, P = “though”
  - 需要多少次比较？

# 简单匹配算法的分析

- 最坏情况:
  - 外层循环: $n - m$
  - 内层循环: $m$
  - 总计 $(n-m)m = O(nm)$
  - 何种输入产生最坏情况?
- 最好情况: $n$-$m$
  - 何时?
- 完全随机的文本和模式:
  - $O(n-m)$

# 提纲

7.1 字符串搜索概述

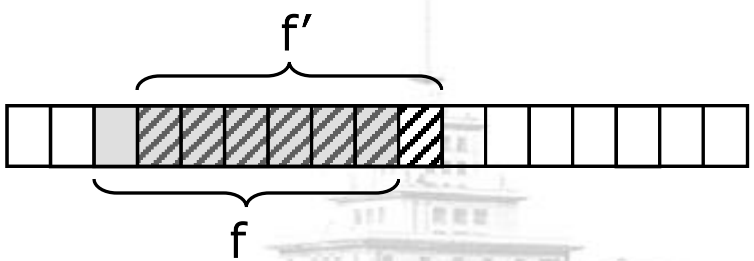
7.2 Rabin-Karp算法

7.3 KMP算法

7.4 BMH算法

# *指纹想法*

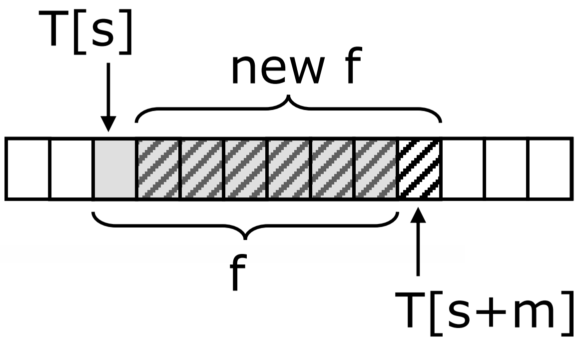
- 假设:
  - 我们可以在O($m$)时间计算一个P的*指纹* $f(P)$.
  - 如果 $f(P) \neq f(T[s \,..\, s+m-1])$, 那么 $P \neq T[s \,..\, s+m-1]$
  - 我们可以在O(1)时间比较指纹
  - 我们可以在O(1)的时间从$f(T[s \,..\, s+m-1])$计算$f' = f(T[s+1 \,..\, s+m])$



# 基于指纹的算法

- 令字母表位 Σ={0,1,2,3,4,5,6,7,8,9}
- 令指纹为一个十进制数, 即, $f(\text{“}1045\text{”}) = 1*10^3 + 0*10^2 + 4*10^1 + 5 = 1045$

```
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
```



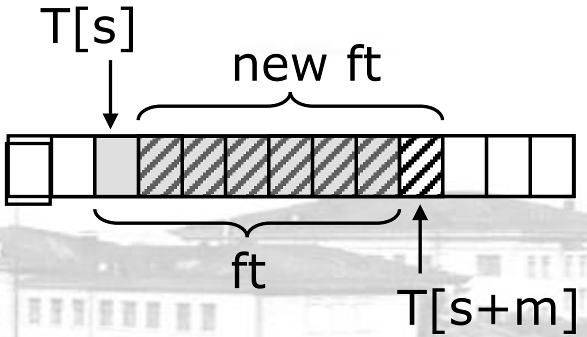

- 运行时间是 2O(m) + O(n–m) = O(n)!

# 使用Hash函数

- 问题: 我们不能假设我们可以对m位数在O(1)时间内进行算术运算
- 解决方案: 使用hash函数 $h = f \bmod q$
  - 例如, 如果 $q = 7$, $h$(“**52**”) = 52 mod 7 = 3
  - $h(S_1) \neq h(S_2) \Rightarrow S_1 \neq S_2$
  - 但 $h(S_1) = h(S_2)$ 不意味着 $S_1=S_2$!
    - 例如, 如果 $q = 7$, h(“**73**”) = 3, 但“**73**” ≠ “**52**”
- 但“mod $q$”算术运算:
  - $(a+b) \bmod q = (a \bmod q + b \bmod q) \bmod q$
  - $(a*b) \bmod q = (a \bmod q)*(b \bmod q) \bmod q$

# 预处理与步骤

- 预处理:
  - $fp = P[m\text{-}1] + 10*(P[m\text{-}2] + 10*(P[m\text{-}3]+ \ldots + 10*(P[1] + 10*P[0])\ldots))$ mod $q$
  - 同样地可以从$T[0..m\text{-}1]$ 计算 $ft$
  - 例如: $P$ = "**2531**", $q$ = 7, $fp$是多少?
- 步骤:
  - $ft = (ft - T[s]*10^{m-1}$ mod $q)*10 + T[s+m])$ mod $q$
  - $10^{m-1}$ mod $q$ 在预处理中计算一次
  - 例: Let $T[\ldots]$ = "**5319**", $q$ = 7, 对应的 $ft$是多少?



# Rabin-Karp算法

```
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
```

- 如果T = “2531978”, P = “1978”需要比较字符多少次?

# 分析

- 如果 $q$ 是素数, hash函数将会使m位字符串在q个值中均匀分配
  – 因此，仅有s个轮换中的每第q次才需要匹配指纹 (匹配需要比较$O(m)$次)
- 期望运行时间 (如果 $q > m$):
  – 预处理: $O(m)$
  – 外循环: $O(n\text{-}m)$
  – 所有内循环: $$\frac{n-m}{q}m = O(n-m)$$
  – 总时间: $O(n\text{-}m)$
- 最坏运行时间: $O(nm)$

# 应用中的Rabin-Karp算法

- 如果字母表有d个字母，将字母翻译为d进制数字，即用d代替算法中的10
- 选择素数$q > m$ 可以利用随机算法在O($m$)时间内完成, 或者$q$ 是一个固定大素数且一个计算机字可以容纳10*$q$.
- Rabin-Karp比较简单，可以容易地拓展到2维模式匹配.

# 提纲

- 7.1 字符串搜索概述
- 7.2 Rabin-Karp算法
- 7.3 KMP算法
- 7.4 BMH算法

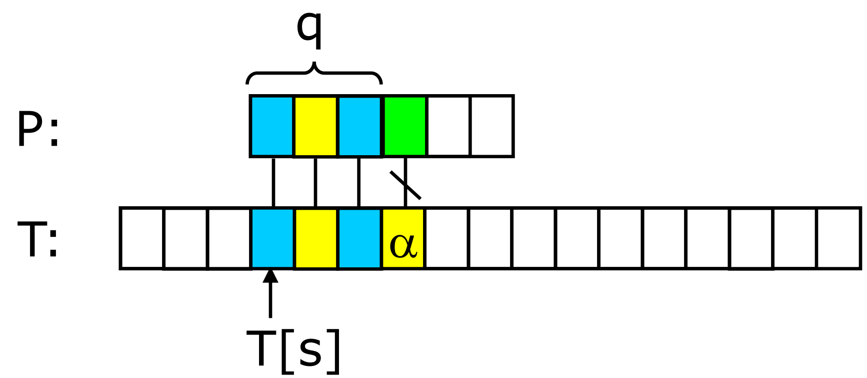
# n次比较的匹配

- 目标:文本中的每个字符仅匹配一次!
- 简单算法的问题:
  - 没有利用已有部分匹配中的知识
  - 例:
    - *T* = "**`Tweedledee and Tweedledum`**" *P* = "**`Tweedledum`**"
    - *T* = "**`pappar`**"

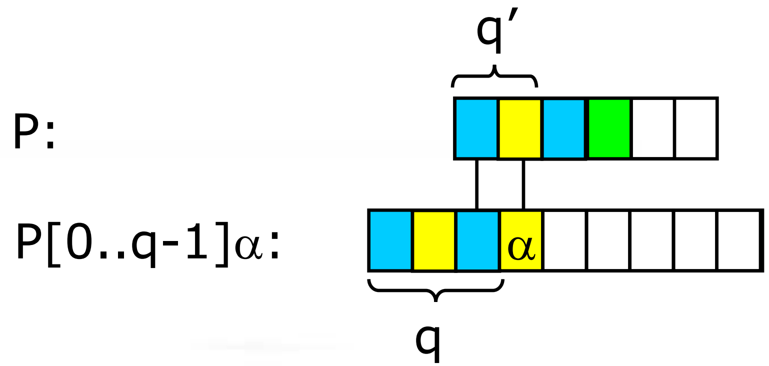
      *P* = "**`pappappappar`**"

# 一般情况

- 算法的状态:
  - 检查移动 *s*,
  - 匹配了 *P* 中的 *q* 个字符
  - 在 *T* 中看到了一个未匹配的字符 $\alpha$.




- 需要寻找:
  - 最大前缀 "*P*-"并且是 $P[1..q-1]\alpha$ 的后缀:
    - $q' = \max\{k \le q \mid P[1..k] = P[q-k+1..q]\alpha\}$

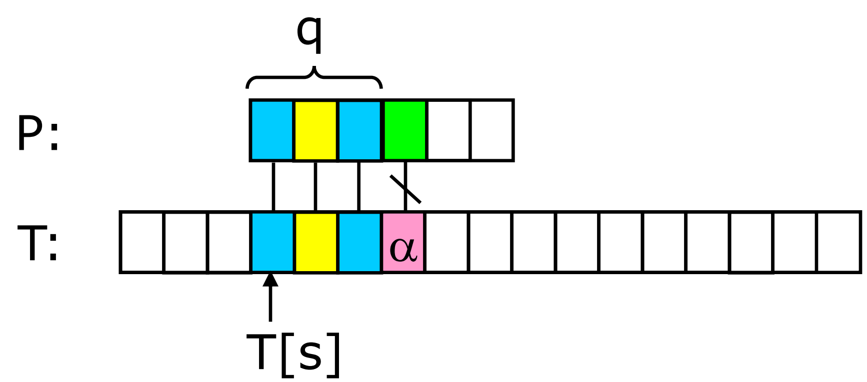


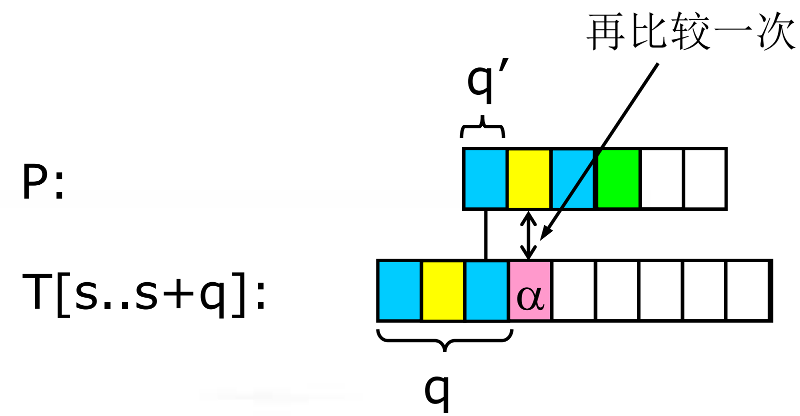
# 自动机搜索

- 算法:
  - 预处理:
    - 对于每个 $q$ $(1 \le q \le m-1)$ 和每个 $\alpha \in \Sigma$ 预先计算一个$q$的新值，记为 $\sigma(q,\alpha)$
    - 填一个大小为$m|\Sigma|$的表
  - 扫描文本
    - 当不匹配发现时 $(P[q] \ne T[s+q])$:
    - 置 $s = s + q - \sigma(q,\alpha) + 1$
    - 分析:
  - ☺ 匹配阶段 $O(n)$
  - ☹ 内存过多: $O(m|\Sigma|)$, 过多的预处理$O(m|\Sigma|)$.

# 前缀函数

- Idea: 忘记未匹配的字符 (α)!
- 算法的状态:
  - 检查变换*s*,
  - 匹配了 *P* 中的*q*个字符
  - 发现了*T*中不匹配的字符α.
- 需要发现:
  - 最大前缀 "*P*-"并且是 *P*[1..*q*-1]的后缀:




- $q' = \pi[q] = \max\{k < q \mid P[1..k] = P[q-k+1..q]\}$

# 前缀表

- 我们可以预先计算大小为m的前缀表来存储$\pi[q]$的值 $(0 \le q < m)$

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

- 计算$P$ = “**dadadu**” 的前缀表

# Knuth-Morris-Pratt 算法

```
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是P上执行KMP算法的本质.**

# KMP的分析

- 最坏运行时间: $O(n+m)$
  - 主算法: $O(n)$
  - **`Compute-Prefix-Table`**: $O(m)$
- 空间: $O(m)$

# 提纲

7.1 字符串搜索概述

7.2 Rabin-Karp算法

7.3 KMP算法

7.4 BMH算法

# 逆简单算法

- 如果从*P*的后面开始搜索？
  - 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
```

- 运行时间和简单算法相同

# 启发式方法

- Boyer和Moore向逆向简单算法中增加了启发式规则，得到了 $O(n+m)$ 算法, 但其更复杂
- Horspool建议仅使用出现启发式规则
  - *在不匹配之后，将*$T[s+m-1]$对齐到模式 $P[0..m-2]$中的最右出现
  - 例:
    - *T*=“**`detective date`**”，*P*=“**`date`**”
    - *T*=“**`tea kettle`**”， *P*=“**`kettle`**”

# 偏移表

- 在预处理中, 计算大小为/Σ|的偏移表.

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

- 例: *P* = “**`kettle`**”
  - shift[**`e`**] =4, shift[**`l`**] =1, shift[**`t`**] =2, shift[**`k`**] =5
- 例: *P* = “**`pappar`**”
  - 其偏移表是什么?

# Boyer-Moore-Horspool 算法

```
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 分析

- 最坏情况运行时间
  - 预处理: $O(|\Sigma|+m)$
  - 搜索: $O(nm)$
    - 何种输入达到此界?
  - 总计: $O(nm)$
- 空间: $O(|\Sigma|)$
  - 和 $m$ 独立
- 在真实数据集合上很快