

算法设计与分析

第九章 字符串匹配算法

哈尔滨工业大学（深圳）

李穆

《算法导论》第32章

串的基本概念

- **串(字符串)**: 是零个或多个字符组成的有限序列。记作: $S = "a_1a_2a_3\dots"$, 其中S是串名, $a_i (1 \leq i \leq n)$ 是单个字符, 可以是字母、数字或其它字符。
- **串值**: 双引号括起来的字符序列是串值。
- **串长**: 串中所包含的字符个数称为该串的长度。
- **空串(空的字符串)**: 长度为零的串称为空串, 它不包含任何字符。

串的基本概念

- **子串(substring):** 串中任意个连续字符组成的子序列称为该串的子串，包含子串的串相应地称为主串。
- **子串的序号:** 将子串在主串中首次出现时的该子串的首字符对应在主串中的序号，称为子串在主串中的序号（或位置）。
- **例如，**设有串A和B分别是：

A=“shenzhenzhen”， B=“zhen”

则B是A的子串， A为主串。B在A中出现了两次，其中首次出现所对应的主串位置是5。因此，称B在A中的序号为5。

- 特别地，空串是任意串的子串，任意串是其自身的子串。

ADT String{

数据对象: $D = \{ a_i | a_i \in \text{CharacterSet}, i=1,2,\dots,n, n \geq 0 \}$

数据关系: $R = \{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i=2,3,\dots,n \}$

基本操作:

StrAssign(t , chars)

StrConcat(s, t)

StrLength(t)

SubString (s, pos, len, sub)

Find(s,t)

}

字符串匹配问题

- 字符串 $T = \text{"at the thought of"}$
- 字符串 $P = \text{"thought"}$

P在T中出现的起始位置下标是7（字符串的首位下标是0），
所以返回 7

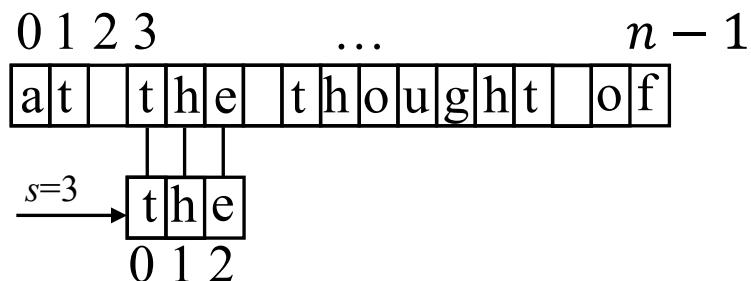
- 字符串 $T = \text{"at the thought of"}$
- 字符串 $P = \text{"think"}$

字符串P在T中不存在，所以返回 -1

字符串T也称为**主串**，字符串P称为**模式串**

字符串匹配问题

- 输入：
 - 长度 n 的主字符串 T 以及长度 m 的模式串 P
- 输出：
 - s – 所有的整数（有效偏移） $(0 \leq s \leq n - m)$ 满足 $T[s, \dots, s + m - 1] = P[0, \dots, m - 1]$ 。如果不存在这样的 s , 则返回 -1



朴素匹配算法

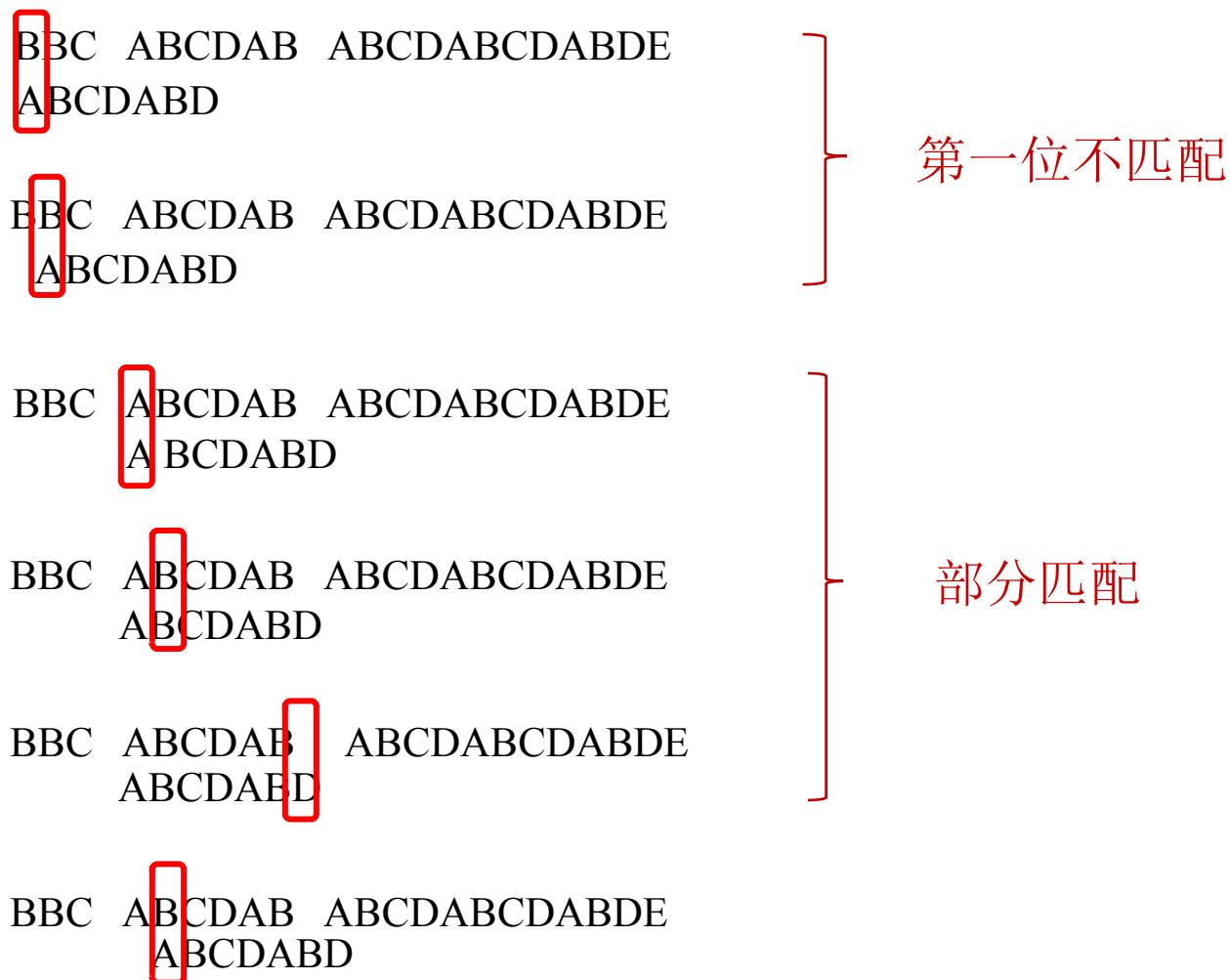


朴素匹配算法

- 朴素的想法: 暴力搜索
 - 直接从头开始, 把主串和模式串的字符逐个匹配, 如果发现不匹配, 再从主串下一位开始

朴素匹配算法示例

- 字符串S: “BBC ABCDAB ABCDABCDABDE”， 模式串P: “ABCDABD”



朴素匹配算法的分析

- 最坏情况：
 - 外层循环: $n - m + 1$
 - 内层循环: m
 - 总计 $(n - m + 1)m = O(nm)$
 - 何种输入产生最坏情况？
例， $T=aaaaaaaa, P=aaa$
- 最好情况: $n - m + 1$
例， $T=aaaaaaab, P=cde$
- 完全随机的文本和模式:
 $O(n - m)$
- 好处是空间要求小

Naive-Search(T,P)

```
1.   n ← length(T);
2.   m ← length(P);
3.   flag ← 1
4.   For s ← 0 to n - m Do
5.       j ← 0 /*模式串P的下标*/
6.       // check if T[s..s+m-1] = P[0..m-1]
7.       While T[s+j] = P[j] Do
8.           j ← j+1
9.       If j == m Then
10.           print s
11.           flag ← 0
12.       If flag Then
13.           Return -1
```

字符串匹配问题

- 输入：
 - 长度 n 的主字符串 T 以及长度 m 的模式串 P
- 输出：
 - s – 所有的整数（有效偏移） $(0 \leq s \leq n - m)$ 满足 $T[s, \dots, s + m - 1] = P[0, \dots, m - 1]$ 。如果不存在这样的 s ，则返回 -1

	0	1	2	3	4	5	6	7	8	9	10	11	12
T	B	B	C		A	B	C	D	A	B		A	B
P	A	B	E	E	E	E	E	E	A	E	E	B	C

朴素匹配算法

0 1 2 3 4 5 6 7 8 9 10 11 12

Naive-Search(T,P)

1. $n \leftarrow \text{length}(T);$
2. $m \leftarrow \text{length}(P);$
3. $\text{flag} \leftarrow 1$
4. For $s \leftarrow 0$ to $n - m$ Do
5. $j \leftarrow 0$
6. While $T[s+j] = P[j]$ Do
7. $j \leftarrow j+1$
8. If $j = m$ Then
9. print s
10. $\text{flag} \leftarrow 0$
11. If flag Then
12. Return -1

B	B	C		A	B	C	D	A	B		A	B
---	---	---	--	---	---	---	---	---	---	--	---	---

$s=4$

A	B	C
---	---	---

$j=j=2$

B	B	C		A	B	C	D	A	B		A	B
---	---	---	--	---	---	---	---	---	---	--	---	---

A	B	C
---	---	---



B	B	C		A	B	C	D	A	B		A	B
---	---	---	--	---	---	---	---	---	---	--	---	---

A	B	C
---	---	---



B	B	C		A	B	C	D	A	B		A	B
---	---	---	--	---	---	---	---	---	---	--	---	---

A	B	C
---	---	---



B	B	C		A	B	C	D	A	B		A	B
---	---	---	--	---	---	---	---	---	---	--	---	---

A	B	C
---	---	---



B	B	C		A	B	C	D	A	B		A	B
---	---	---	--	---	---	---	---	---	---	--	---	---

A	B	C
---	---	---



B	B	C		A	B	C	D	A	B		A	B
---	---	---	--	---	---	---	---	---	---	--	---	---

A	B	C
---	---	---



A	A	A	A	A	A	B	A	A	A	A	A	B
---	---	---	---	---	---	---	---	---	---	---	---	---

A	A	A	B
---	---	---	---

A	A	A	A	A	A	B	A	A	A	A	A	B
---	---	---	---	---	---	---	---	---	---	---	---	---

A	A	A	B
---	---	---	---

A	A	A	A	A	A	B	A	A	A	A	A	B
---	---	---	---	---	---	---	---	---	---	---	---	---

A	A	A	B
---	---	---	---

A	A	A	A	A	A	B	A	A	A	A	A	B
---	---	---	---	---	---	---	---	---	---	---	---	---

A	A	A	B
---	---	---	---

A	A	A	A	A	A	B	A	A	A	A	A	B
---	---	---	---	---	---	---	---	---	---	---	---	---

A	A	A	B
---	---	---	---

A	A	A	A	A	A	B	A	A	A	A	A	B
---	---	---	---	---	---	---	---	---	---	---	---	---

A	A	A	B
---	---	---	---

朴素匹配算法的分析

- 朴素匹配算法的**效率低**，是因为在匹配过程中，它需要比对文本和模式串的每个字符。
- 前一次匹配的信息完全被扔掉，后一次匹配时，需从头再来。
- 完全忽略了模式串 P 的自身组成特点。

思路一：模式串看成整体， $O(1)$ 判断出来当前位置是否匹配

A	A	A	A	A	A	B	A	A	A	A	A	A	B
---	---	---	---	---	---	---	---	---	---	---	---	---	---

AAAB

A	A	A	A	A	A	B	A	A	A	A	A	A	B
---	---	---	---	---	---	---	---	---	---	---	---	---	---

AAAB

思路二：主串不回退，“从哪里跌倒就从哪里站起来”



A	A	A	A	A	A	B	A	A	A	A	A	A	B
---	---	---	---	---	---	---	---	---	---	---	---	---	---

A	A	A	A	B
---	---	---	---	---

A	A	A	A	A	A	B	A	A	A	A	A	A	B
---	---	---	---	---	---	---	---	---	---	---	---	---	---

A	A	A	B
---	---	---	---

基于指纹的匹配算法

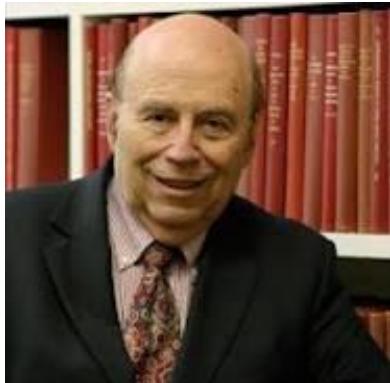


基于指纹的算法

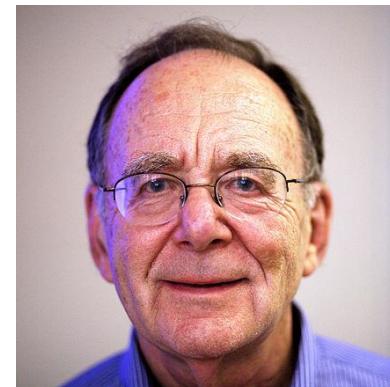
- 为了避免挨个字符，对主串和模式串进行比较，能否一次性判断模式串与主串中的子串是否相等？

Rabin-Karp 算法

- 由Michael O. Rabin和Richard M. Karp在1987年提出



Michael Oser Rabin, 以色列
计算机科学家, 1976年图灵奖
得主

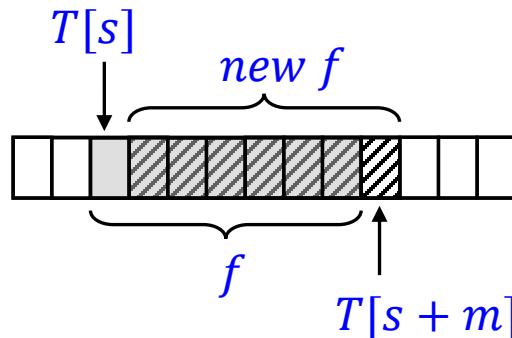


Richard Manning Karp, 计算
机科学家, 1985年的图灵奖得
主

基于指纹的算法

- 令字母表 $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$
- 主字符串 $T = "921045"$, 模式串 $P = "1045"$ 令指纹为一个十进制数, 即

$$\begin{aligned}f("1045") &= 1 * 10^3 + 0 * 10^2 + 4 * 10^1 + 5 \\&= 1045\end{aligned}$$



基于指纹的算法

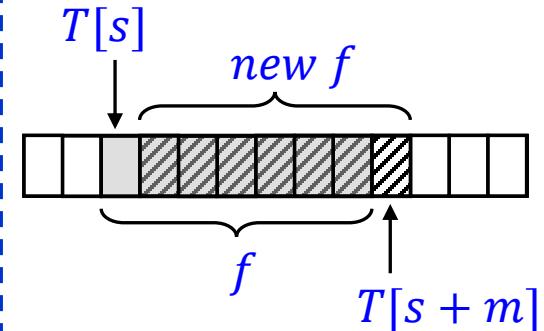
$T = "921045"$
 $P = "1045"$

- 可以在 $O(m)$ 时间计算一个 P 的指纹 $f(P) = 1045$, $f(T[0, \dots, 3]) = 9210$
- 如果 $f(P) \neq f(T[s, \dots, s + m - 1])$, 那么 $P \neq T[s, \dots, s + m - 1]$, 我们可以在 $O(1)$ 时间比较指纹
- 我们可以在 $O(1)$ 的时间从 $f(T[s, \dots, s + m - 1])$ 计算 $f(T[s + 1, \dots, s + m])$
 - $f(T[s + 1, \dots, s + m]) = (f(T[s, \dots, s + m - 1]) - T[s] * 10^{m-1}) * 10 + T[s + m]$
 - 例 $f(T[1, \dots, 4]) = (9210 - 9000) * 10 + 4 = 2104$

基于指纹的算法

Fingerprint-Search(T, P)

```
1. fp  $\leftarrow$  compute  $f(P)$  /*计算模式串P的指纹*/
2. f  $\leftarrow$  compute  $f(T[0..m-1])$  /*主串T前m的指纹*/
3. flag  $\leftarrow$  1
4. For s = 0 To n - m Do
    If fp = f Then
        print s
        flag  $\leftarrow$  0
    f  $\leftarrow$  (f - T[s]* $10^{m-1}$ )*10 + T[s+m]
9. If flag Then
10. return -1
```



运行时间是 $2O(m) + O(n - m) = O(n)$

基于指纹算法的缺陷

- 当模式串 P 过长时，即 m 过大，对应的数值 $f(P)$ 过大，会导致溢出
- 用数组模拟大数，存储指纹，当两个过大的数值比较大小时，需要遍历两个数组，这样两数比较，我们不能假设可以在 $O(1)$ 时间内完成

Rabin-Karp算法



使用Hash函数

- 解决方案： 使用hash函数 $h = f \bmod q$
 - 例如，如果 $q = 7, h("52") = 52 \bmod 7 = 3$
 - 这样指纹的值不会大于 q ，限制了需要比较的数值的范围
 - $h(S_1) \neq h(S_2) \Rightarrow S_1 \neq S_2$
 - 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” \neq “52”

当 $h(S_1) = h(S_2)$ 时，需要把 $T[s + 1, \dots, s + m]$ 和 $P[1, \dots, m]$ 这两个字符串逐个字符比较，每个字符都一样，才能最终断定 $T[s + 1, \dots, s + m] = P[1, \dots, m]$

“mod q” 算术运算

- 公式1:

$$(a \pm b) \bmod q = (a \bmod q \pm b \bmod q) \bmod q$$

- 公式2:

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

预处理与步骤

- 预处理:
 - $fp = (P[m - 1] + 10 * (P[m - 2] + 10 * (P[m - 3] + \dots + 10 * (P[1] + 10 * P[0]) \dots)) \bmod q$
 - 同样地可以从 $T[0, \dots, m - 1]$ 计算 ft
 - 例如: $P = "2531"$, $q = 13$, fp 是多少?

基于公式 $f(T[s + 1, \dots, s + m]) = (f(T[s, \dots, s + m - 1]) - T[s] * 10^{m-1}) * 10 + T[s + m]$, 套用 “**mod q**” 算术运算公式1, 2 得到

- 步骤:
 - $ft \leftarrow ((ft - T[s] * 10^{m-1} \bmod q) * 10 + T[s + m]) \bmod q$, 一般取 q 大于任意的 $T[i]$ 的素数, 当然 q 大于 10
 - $10^{m-1} \bmod q$ 在预处理中计算一次

Rabin-Karp算法

Rabin-Karp-Search(T,P)

```
1. q ← a /*q取为素数*/  
2. c ←  $10^{m-1} \text{ mod } q$   
3. fp ← 0, ft ← 0, flag ← 1  
4. For i = 0 To m - 1 Do // preprocessing  
    fp ← (10*fp + P[i]) mod q  
    ft ← (10*ft + T[i]) mod q  
5. 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  
            print s  
        flag ← 0  
    ft ← ((ft - T[s]*c)*10 + T[s+m]) mod q  
13. If flag Then  
14. Return -1
```

套用mod q运算
公式1,2得到

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
 - 从概率上讲，每 q 次才会遇到指纹匹配（需要每个字符进行比较，时间复杂度为 $O(m)$ ）
- 期望运行时间（如果 $q > m$ ）：
 - 预处理: $O(m)$ (4--6行)
 - 外循环: $O(n - m)$ (第7行)
 - 所有内循环: $\frac{n-m}{q}m = O(n - m)$ (第9行)
 - 总时间: $O(n - m)$

Rabin-Karp-Search(T,P)

```
1.   q ← a /*q取为素数*/
2.   c ←  $10^{m-1} \text{ mod } q$ 
3.   fp ← 0, ft ← 0, flag ← 1
4.   for i = 0 to m-1 // preprocessing
5.       fp ← (10*fp + P[i]) mod q
6.       ft ← (10*ft + T[i]) mod q
7.   for s = 0 to n - m // matching
8.       if fp = ft then //run a loop to compare strings
9.           if P[0..m-1] = T[s..s+m-1] then
10.              print s
11.              flag ← 0
12.              ft ← ((ft - T[s]*c)*10 + T[s+m]) mod q
13.           if flag then
14.               return -1
```

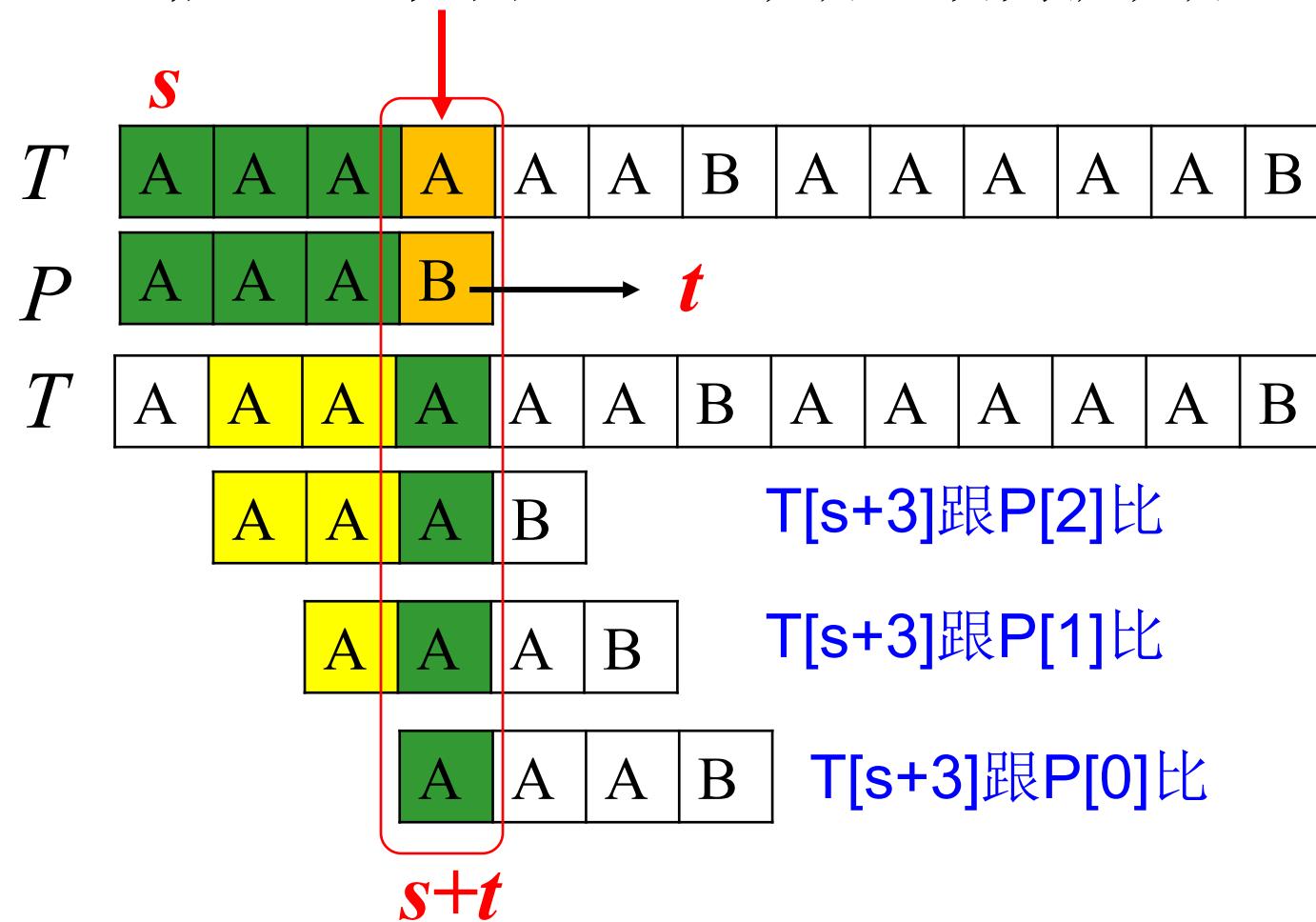
分析

- 最坏运行时间： $O(nm)$
何时？
- 前一次匹配的信息其实有部分可以应用到后一次匹配中去，而朴素的字符串匹配算法把这个信息扔掉了，Rabin-Karp算法通过指纹的思想，对其进行了很好的利用

应用中的Rabin-Karp算法

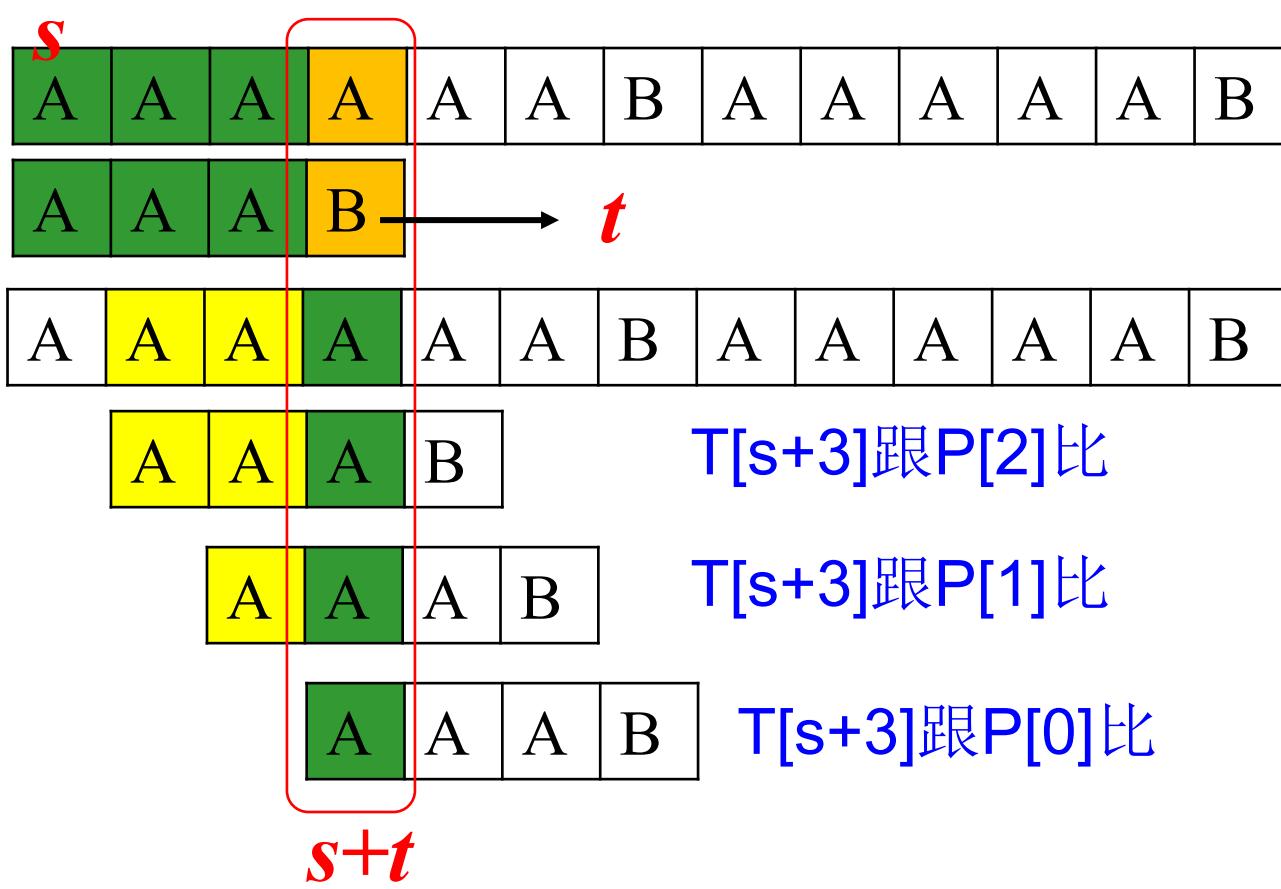
- 如果字母表有 d 个字母， 将字母翻译为 d 进制数字， 即用 d 代替算法中的10
- Rabin-Karp比较简单， 可以容易地拓展到2维模式匹配， 以及多模式匹配问题。
- 虽然在理论上并不比朴素匹配算法更优， 但在实际应用中优势明显。
- 如果能够选择一个好的哈希函数， 它的效率将会很高， 而且也易于实现。

思路二：主串不回退，“从哪里跌倒就从哪里站起来”

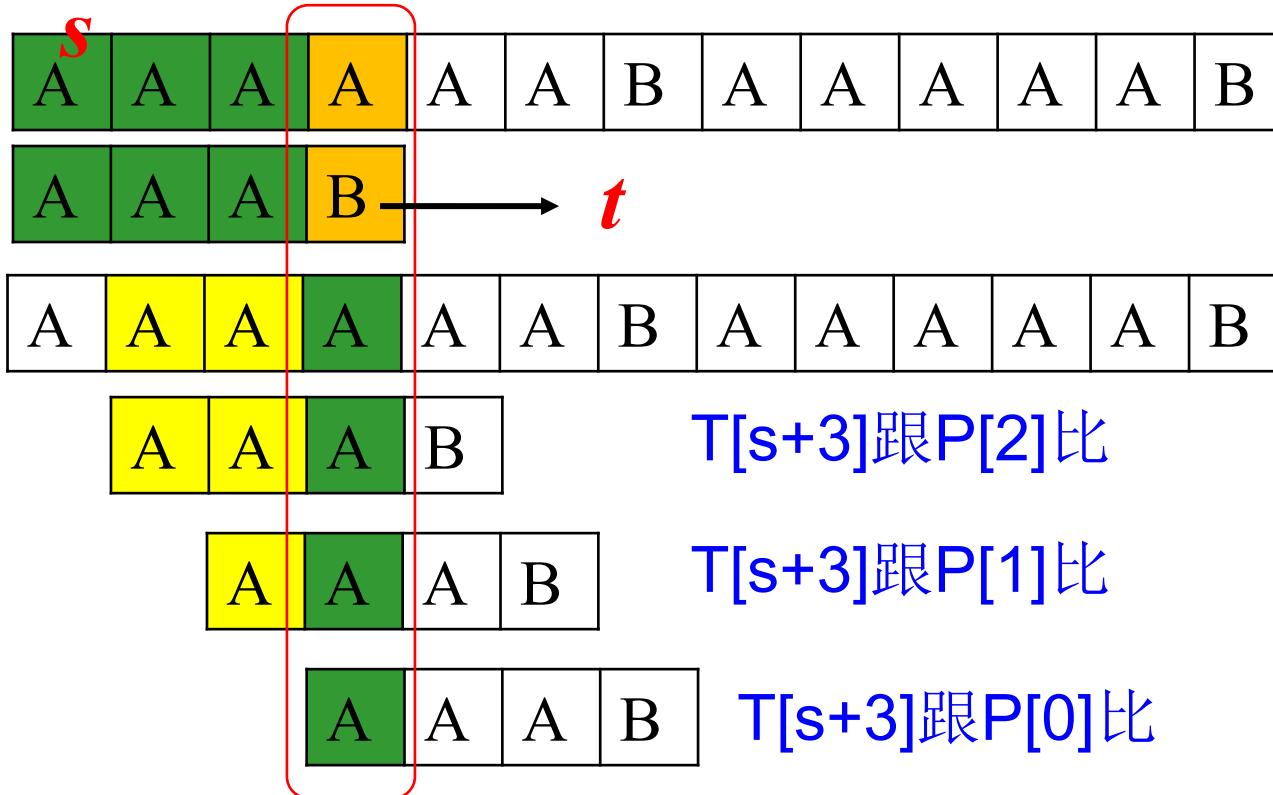


$T[s+t]$ 与失配的字符 $P[t]$ 之前的字符比，与 $P[k]$ 比，表示？

表示从 $s+t-k$ 处开始进行 P 和 T 的比对， $0 \leq k < t$ ，代表从遍历 $s+1$ 到 $s+t$ 的所有情况



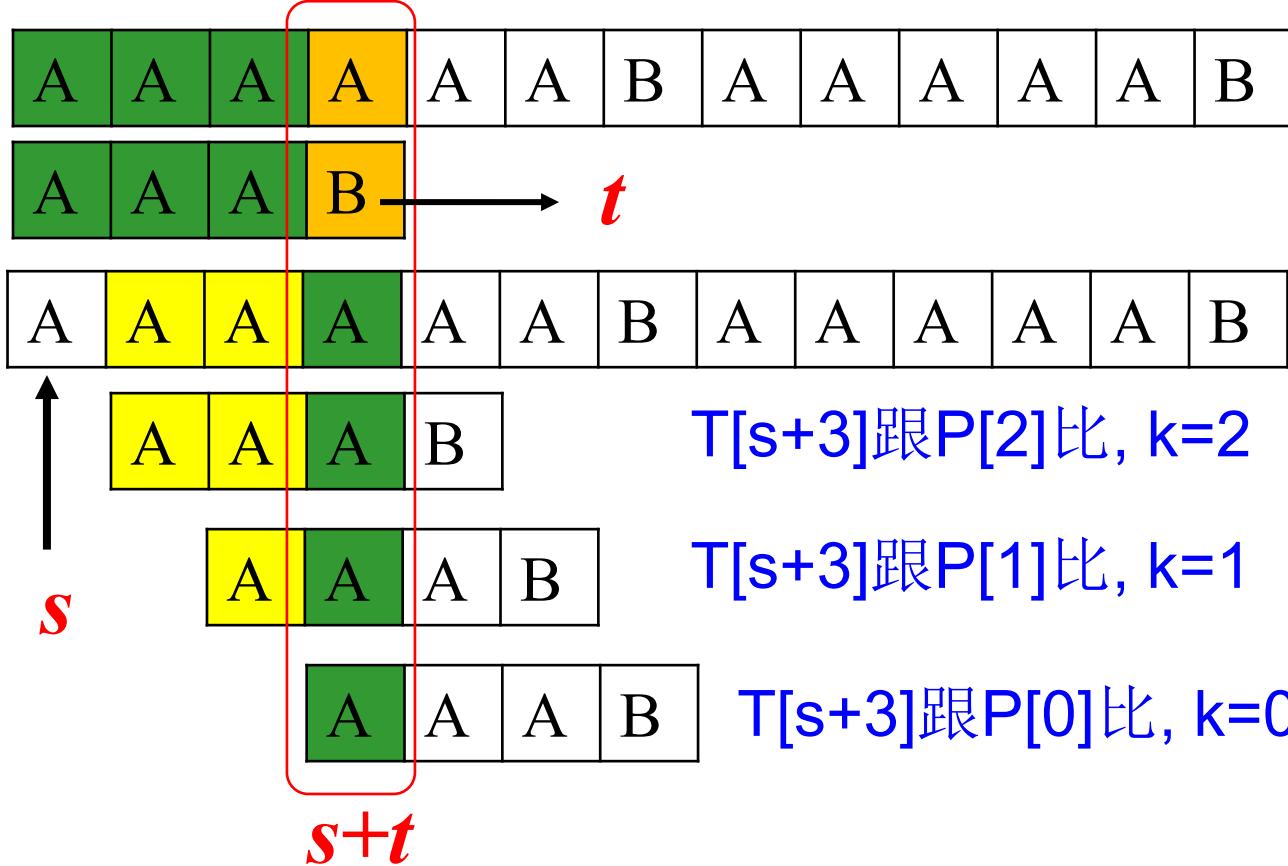
- $T[s+t]$ 与 $P[k]$ 比， 隐含着： $T[s+t-k:s+t-1]$ 与 $P[0:k-1]$ 匹配
只需验证匹配的情况，如果不匹配，不可能在 $s+t-k$ 处的 T 与 P 的匹配。
- 怎么判断匹配不匹配？



$s+t$

找k，让 $T[s+t-k:s+t-1]$ 与 $P[0:k-1]$ 匹配

- T和P是在 $T[s+t]$ 和 $P[t]$ 处失配的，在此之前， $P[0:t-1]$ 与 $T[s:s+t-1]$ 匹配
- $T[s+t-k:s+t-1] = P[t-k:t-1]$
- 只需找 $P[0:k-1] = P[t-k:t-1]$ 的情况
- $P[t-k:t-1]$ 是字符串 $P[0:t-1]$ 的后缀， $P[0:k-1]$ 是P的前缀



- 在 $P[t]$ 处失配
- 只需找 $P[0:k-1]$ 是 $P[0:t-1]$ 后缀的情况
- 如果有多个符合条件的 k , 怎么办?
- 找 k 最大的情况, 对应着离当前失配的位置 s 最近的可能出现配对的位置

思路二：主串不回退，“从哪里跌倒就从哪里站起来”

原地踏步

A	B	A	B	D	A	B	A	B	A	A	A	B
---	---	---	---	---	---	---	---	---	---	---	---	---

A	B	A	B	C
---	---	---	---	---

A	B	A	B	C
---	---	---	---	---

A	B	A	B	C
---	---	---	---	---

A	B	A	B	C
---	---	---	---	---

爬起来直接往前走

A	B	A	B	D	A	B	A	B	A	A	A	B
---	---	---	---	---	---	---	---	---	---	---	---	---

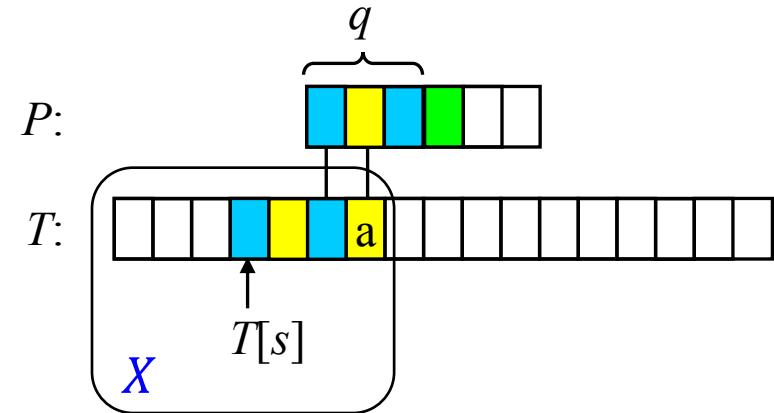
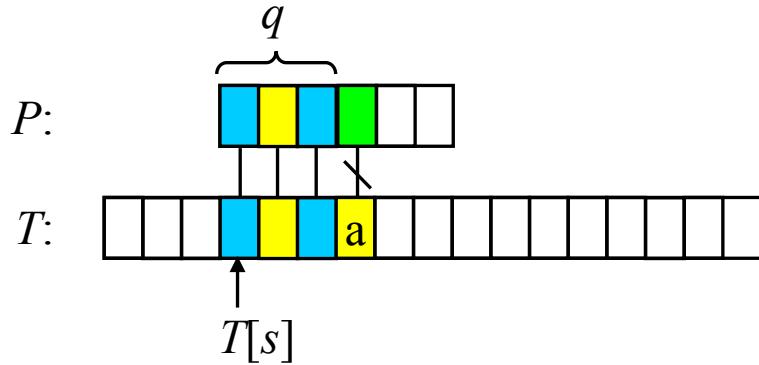
A	B	A	B	C
---	---	---	---	---

A	B	A	B	C
---	---	---	---	---

有限状态自动机 字符串匹配算法



有限自动机字符串匹配算法



- 我们能否设法利用这个已知信息，不要把"搜索位置"移回已经比较过的位置，而是继续把它向后移？
- 固定主串 T ，移动模式串 P ，并记录已检验的字符串 X 。当 X 的后缀和 P 的前缀相同字符等于 P 的长度时，匹配成功

例子

字符串: $T = "cbababacaba"$
模式串: $P = "ababaca"$

考慮以下过程:

- 每一步读入 T 的一个字符，用 X 记录已读入的的字符，
- 同时，记录满足以下条件的最长字符串 S 以及其长度 K : S 是 X 的后缀，同时也是 P 的前缀

第1步: $X = c, S_1 = "", K_1 = 0$

$$K_i = \sigma(S_i) = \max\{k: P[0, k - 1] \text{是} X \text{的后缀}\}$$

第2步: $X = cb, S_2 = "", K_2 = 0$



第3步: $X = cba, S_3 = P[0] = "a", K_3 = 1$



第4步: $X = cbab, S_4 = P[0,1] = "ab", K_4 = 2$



第5步: $X = cbaba, S_5 = P[0,1,2] = "aba", K_5 = 3$

第6步: $X = cbabab, S_6 = P[0,1,2,3] = "abab", K_6 = 4$

第7步: $X = cbababa, S_7 = P[0,1,2,3,4] = "ababa", K_7 = 5$

第8步: $X = cbababac, S_8 = P[0,1,2,3,4,5] = "ababac", K_8 = 6$

第9步: $X = cbababaca, S_9 = P[0,1,2,3,4,5,6] = "ababaca", K_9 = 7$

第10步: $X = cbababacab, S_{10} = P[0,1] = "ab", K_{10} = 2$

第11步: $X = cbababacaba, S_{11} = P[0,1,2] = "aba", K_{11} = 3$

换个角度

字符串: $T = "cbabababacaba"$
模式串: $P = "ababaca"$

第1步: $X = c, S_1 = "", K_1 = 0$

第2步: $X = cb, S_2 = "", K_2 = 0$

第3步: $X = cba, S_3 = P[0] = "a", K_3 = 1$

第4步: $X = cbab, S_4 = P[0,1] = "ab", K_4 = 2$

第5步: $X = cbaba, S_5 = P[0,1,2] = "aba", K_5 = 3$

第6步: $X = cbabab, S_6 = P[0,1,2,3] = "abab", K_6 = 4$

第7步: $X = cbababa, S_7 = P[0,1,2,3,4] = "ababa", K_7 = 5$

第8步: $X = cbababac, S_8 = P[0,1,2,3,4,5] = "ababac", K_8 = 6$

第9步: $X = cbababaca, S_9 = P[0,1,2,3,4,5,6] = "ababaca", K_9 = 7$

第10步: $X = cbababacab, S_{10} = P[0,1] = "ab", K_{10} = 2$

第11步: $X = cbababacaba, S_{11} = P[0,1,2] = "aba", K_{11} = 3$

分析:

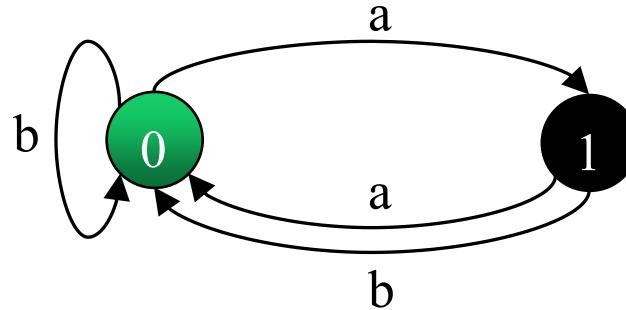
1、需将主串 T 的字符都读入, $O(n)$

2、每步需比对 P 的前缀与 X 的后缀, 最坏情况下: $O(m)$

3、最坏情况下 $O(nm)$

有没有办法, 不用每个比对, 就知道 P 的哪些前缀可以构成 X 的后缀?

有限状态自动机



有限状态自动机 M 是一个五元组, $M = \{Q, q_0, A, \Sigma, \delta\}$:

- Q 是状态的**有限集合**, 即状态自动机中所有可能出现的转移状态。
- q_0 是有限状态自动机的起始状态, $q_0 \in Q$ 。
- $A \subseteq Q$ 是一个接受状态集合, 它是 Q 的一个**子集**, 通常一个字符串输入完毕后, 如果当前的状态是 A 中的一个元素, 则表示该字符串被接受, 否则表示被拒绝。
- Σ 所有可能的输入字母表集合。
- δ 被称为 M 的状态转移函数, $Q \times \Sigma \rightarrow Q$ 的一个映射函数。

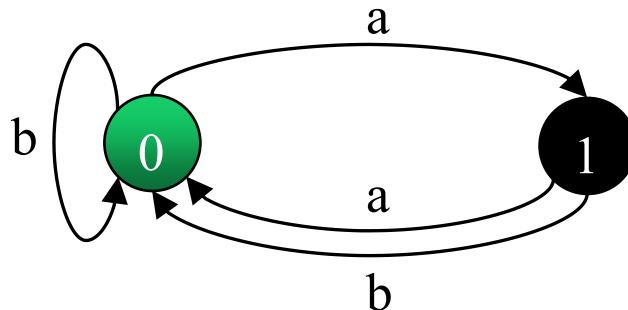
有限状态自动机

状态集合: $Q = \{0,1\}$

初始状态: $q_0 = 0$

接受状态: $A = \{1\}$

字母集合: $\Sigma = \{a, b\}$



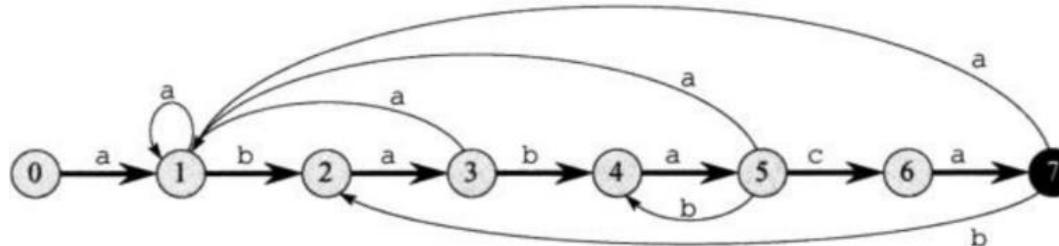
状态转移函数 $\delta =$

状态	输入	
	a	b
0	1	0
1	0	0

- 给定字符串 $abaaa$, 状态的变化序列为?
 - $\{0, 1, 0, 1, 0, 1\}$, 由于最后状态处于状态 **1**, 该字符串可以被状态机接受
- 给定字符串 $abbaa$, 状态的变化序列为?
 - $\{0, 1, 0, 0, 1, 0\}$, 由于最后状态处于状态 **0**, 该字符串被状态机拒绝

有限状态自动机字符串匹配算法

主字符串: $T = "abababacaba"$ 模式串: $P = "ababaca"$



状态集合: $Q = \{0, 1, 2, 3, 4, 5, 6, 7\}$, 模式串 P 与主字符串 T 当前已匹配上的字符个数

初始状态: $q_0 = 0$

接受状态: $A = \{7\}$, 当状态 q 属于 A , 表示该字符串被接受

字母集合: $\Sigma = \{'a', 'b', 'c'\}$

状态转移函数: $\delta(q, x) = \sigma(P[0, q - 1] \ x) = \max \{k: P[0, k - 1] \text{ 是 } P[0, q - 1] + x \text{ 的后缀}\}$

(状态转移函数就是刻画新加入主串 T 的字符 x , 使得匹配状态发生的具体变化 (指的是匹配字符个数)。具体是: $q \in Q$ 是目前状态, 也表示已找到的模式串 P 与主串 T 已匹配字符的个数, 即 $P[0, q - 1]$; 当前已输入主串 T 所有字符与模式串 P 匹配的字符个数等于 $P[0, k - 1]$ 是新字符串 $P[0, q - 1] + 'x'$ 的后缀个数)

第5步: $X = ababa, S_5 = P[0, 1, 2, 3, 4] = "ababa", \delta(4, a) = 5$

第6步: $X = ababab, S_6 = P[0, 1, 2, 3] = "abab", \delta(5, b) = 4$

有限状态自动机字符串匹配算法

例：主字符串： $T = "cbabababacaba"$ 模式串： $P = "ababaca"$

$P:$

a	b	a	b	a	c	a
---	---	---	---	---	---	---

$T:$

c	b	a	b	a	b	a	b	c	a	b	a							
---	---	---	---	---	---	---	---	---	---	---	---	--	--	--	--	--	--	--

第1步： $X = c, q = 0, \delta(q, x) = \delta(0, c) = 0$

第2步： $X = cb, q = 0, \delta(q, x) = \delta(0, b) = 0$

第3步： $X = cba, q = 0, \delta(q, x) = \delta(0, a) = 1$

第4步： $X = cbab, q = 1, \delta(q, x) = \delta(1, b) = 2$

字符串 X 的后缀和模式串 P 的前缀 相匹配的字符个数

等于 $P[0, q - 1] + 'x'$ 的后缀和模式串 P 的前缀相匹配的字符个数

第5步： $X = cbaba, q = 2, \delta(q, x) = \delta(2, a) = 3$

第6步： $X = cbabab, q = 3, \delta(q, x) = \delta(3, b) = 4$

第7步： $X = cbababa, q = 4, \delta(q, x) = \delta(4, a) = 5$

第8步： $X = cbababab, q = 5, \delta(q, x) = \delta(5, b) = 4$

状态转移函数： $\delta(q, x) = \sigma(P[0, q - 1]'x') = \max \{k : P[0, k - 1]$
是 $P[0, q - 1] + 'x'$ 的后缀}

$P[0, q - 1] + 'x' = P[0, 0] + 'b' = ab$

$P[0, q - 1] + 'x' = P[0, 1] + 'a' = aba$

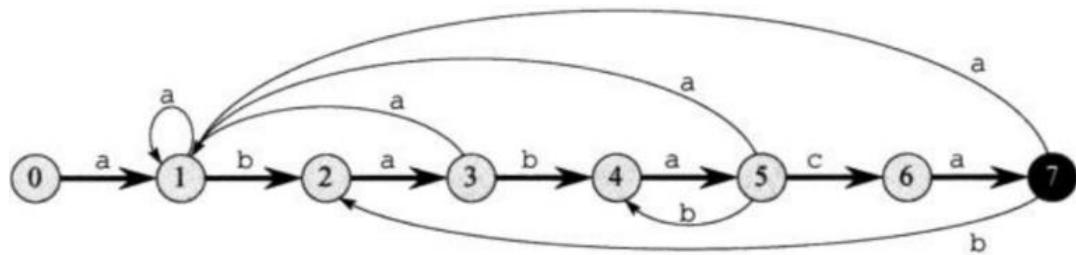
$P[0, q - 1] + 'x' = P[0, 2] + 'b' = abab$

$P[0, q - 1] + 'x' = P[0, 3] + 'a' = ababa$

$P[0, q - 1] + 'x' = P[0, 4] + 'b' = ababab$

有限状态自动机字符串匹配算法

主字符串: $T = "abababacaba"$ 模式串: $P = "ababaca"$



- $\delta(q, x) = \sigma(P[0, q - 1]'x') = \max\{k: P[0, k] \text{ 是 } P[0, q - 1] + 'x' \text{ 的后缀}\}$
- 状态转移函数可以表示为一个 $(m + 1) \times |\Sigma|$ 的表

$$\delta =$$

输入 状态 \	'a'	'b'	'c'
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	4	6
6	7	0	0
7	1	2	0

构造状态转移表

Compute-Transition-Table(P, S)

1. $m \leftarrow P.length$ /*模式串P的长度*/
2. for $q = 0$ to m /* q 已匹配字符个数*/
3. for each character x in S /*考虑每个可能加入的字母*/
4. $k \leftarrow \min(m, q+1)$
5. repeat
6. $k \leftarrow k-1$
7. until $P[0:k]$ is the suffix of $P[0:q-1]+'x'$
8. $\delta(q, 'x') \leftarrow k+1$
9. return δ

每次状态变化， $\delta(q, x)$ 最大是 $(q + 1)$ 或者 m

最坏时间复杂度 $O(m^3|\Sigma|)$ ，可以改进为 $O(m|\Sigma|)$

有限状态自动机字符串匹配算法

Finite-Automation-Matcher(T, P, Σ)

```
1. n ← T.length
2. q ← 0, flag ← 1
3. δ ← Compute-Transition-Table(P, S) /*构造状态转移表*/
4. For i = 0 To n Do /*对主串T的每个字符*/
    5.     q ← δ(q, T[i])
    6.     If q = m Then
        7.         Print “pattern occurs with shift” i-m+1      /* i-m+1匹配开始的
           下标*/
    8.     flag ← 0
    9.     If flag Then
10.        Return -1
```

- 分析:

- 匹配阶段: $O(n)$
- 内存过多: $O(m|\Sigma|)$, 过多的预处理 $O(m^3|\Sigma|)$

思路二：主串不回退，“从哪里跌倒就从哪里站起来”

爬起来直接往前走

A	B	A	A	B	A	C	A	B	A	A	A	B
---	---	---	---	---	---	---	---	---	---	---	---	---

A	B	A	C
---	---	---	---

A	B	A	C
---	---	---	---

内存过多： $O(m|\Sigma|)$, 过多的预处理 $O(m^3|\Sigma|)$

原地踏步

A	B	A	A	B	A	C	A	B	A	A	A	B
---	---	---	---	---	---	---	---	---	---	---	---	---

A	B	A	C
---	---	---	---

A	B	A	C
---	---	---	---

A	B	A	C
---	---	---	---

忽略当前要匹配的字符' x '，重点利用已匹配字符 $P[0, \dots, q - 1]$ 的信息

KMP 算法



前缀表 π

- 前缀函数(最长相等前后缀):

$\pi[q] = \max\{k < q \mid P[0, \dots, k-1] = P[q-k, \dots, q-1]\}$ 即
 $\pi[q] = \max\{k\}$, 当模式串 $P[0, \dots, q-1]$ 的前缀等于当模式串 $P[0, \dots, q-1]$ 的长度为 k 的后缀。

- q : 模式串 P 与主字符串 T 中已匹配上的字符个数
- 给定模式串 $P = "ABCDABD"$, 计算 $\pi[5]$:
 - $P[0, \dots, 4] = 'ABCDA'$
 - 前缀包括, $A, AB, ABC, ABCD$
 - 后缀包括, $A, DA, CDA, BCDA$
 - 因此 $\pi[5] = 1$

0	1	2	3	4	5	6	7	
$\pi[q]$	0	0	0	0	0	1	2	0

预先构造前缀表 π

Compute-Prefix(P)

1. $m \leftarrow P.length$
2. Let $\pi[0, \dots, m]$ be a new array
3. $\pi[0] \leftarrow 0, \pi[1] \leftarrow 0$
4. $k \leftarrow 0$
5. For $q = 2$ To m Do
6. While $k > 0$ and $P[k] \neq P[q-1]$
7. $k = \pi[k]$
8. If $P[k] == P[q-1]$
9. $k \leftarrow k + 1$
10. $\pi[q] \leftarrow k$
11. Return π

q : 模式串 P 与主字符串 T 中已匹配上的字符个数

时间复杂度 $\Theta(m)$

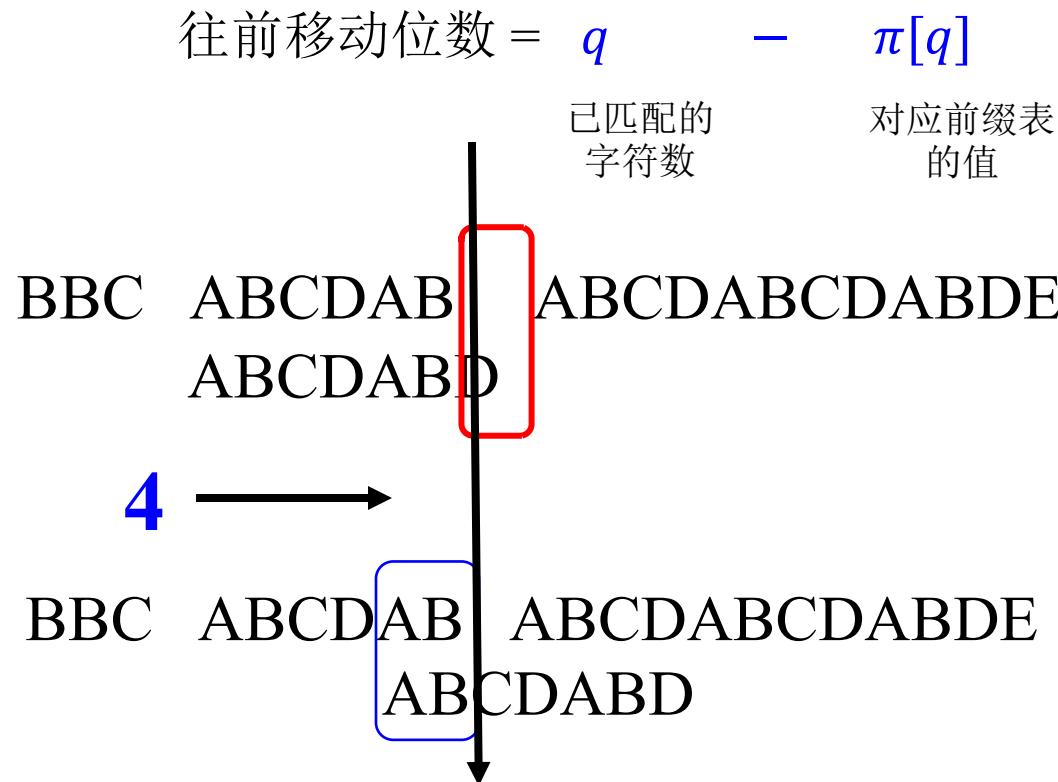
Knuth-Morris-Pratt 算法

KMP-Search(T,P)

```
1. m ← P.length
2. n ← T.length, flag ← 1
3.  $\pi$  ← Compute-Prefix(P)
4. q ← 0 // number of characters matched
5. For i = 0 To n-1 // scan the text from left to right
   While q > 0 and P[q] ≠ T[i]//next character doesn't match
   q ←  $\pi$  [q]
8. If P[q] = T[i]//next character matches
9. q ← q + 1
10. If q = m //all P's characters matched
11.   Print "pattern occurs with shift" i-m+1,
12.   flag ← 0
13.   q ←  $\pi$  [q]
14. If flag Then
15.   Return -1
```

前缀表 π 到底存了什么？

前缀表 π 中实际存了针对当前部分匹配的 q 个字符，**末尾有多少个字符有可能属于下一个完全匹配**，从而快速跳过那些完全没有可能属于下一个完全匹配的字符。



部分匹配 $q = 6$ ，可能
有用 $\pi[6] = 2$ ，
因此，前移 $6 - 2 = 4$

KMP的分析

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

KMP vs 有限状态自动机

KMP-Search(T,P)

```
1. m ← P.length  
2. n ← T.length,flag ← 1  
3.  $\pi$  ← Compute-Prefix(P)  
4. q ← 0  
5. For i = 0 To n-1 Do  
    While q > 0 and P[q] ≠ T[i]  
        q ←  $\pi$ [q]  
    If P[q] = T[i]  
        q ← q + 1  
10. If q = m  
    Print "pattern occurs with shift" i-m+1, flag ← 0  
12. q ←  $\pi$ [q]  
13. If flag Then  
    Return -1
```

Finite-Automation-Matcher(T,P, Σ)

```
1. n ← T.length  
2. q ← 0,flag ← 1  
3.  $\delta$  ← Compute-Transition-Table(P, S)  
4. For i = 0 To n Do /*对主串T的每个字符*/  
    q ←  $\delta$ (q,T[i])  
6. If q = m Then  
    Print "pattern occurs with shift" i-m+1  
8. flag ← 0  
9. If flag Then  
10. Return -1
```

- 有限状态自动机将所有可能的状态转移预先进行计算，存储在 δ 表中
- KMP使用数组 π 即时有效的计算状态转移，避免大量的无用计算和存储

Knuth-Morris-Pratt 算法

The algorithm was conceived by James H. Morris and independently discovered by Donald Knuth “a few weeks later” from automata theory. Morris and Vaughan Pratt published a technical report in 1970. The three also published the algorithm jointly in 1977. Independently, in 1969, Matiyasevich discovered a similar algorithm, coded by a two-dimensional Turing machine, while studying a string-pattern-matching recognition problem over a binary alphabet. **This was the first linear-time algorithm for string matching.**——WikiPedia



Donald Ervin Knuth
(1938-)

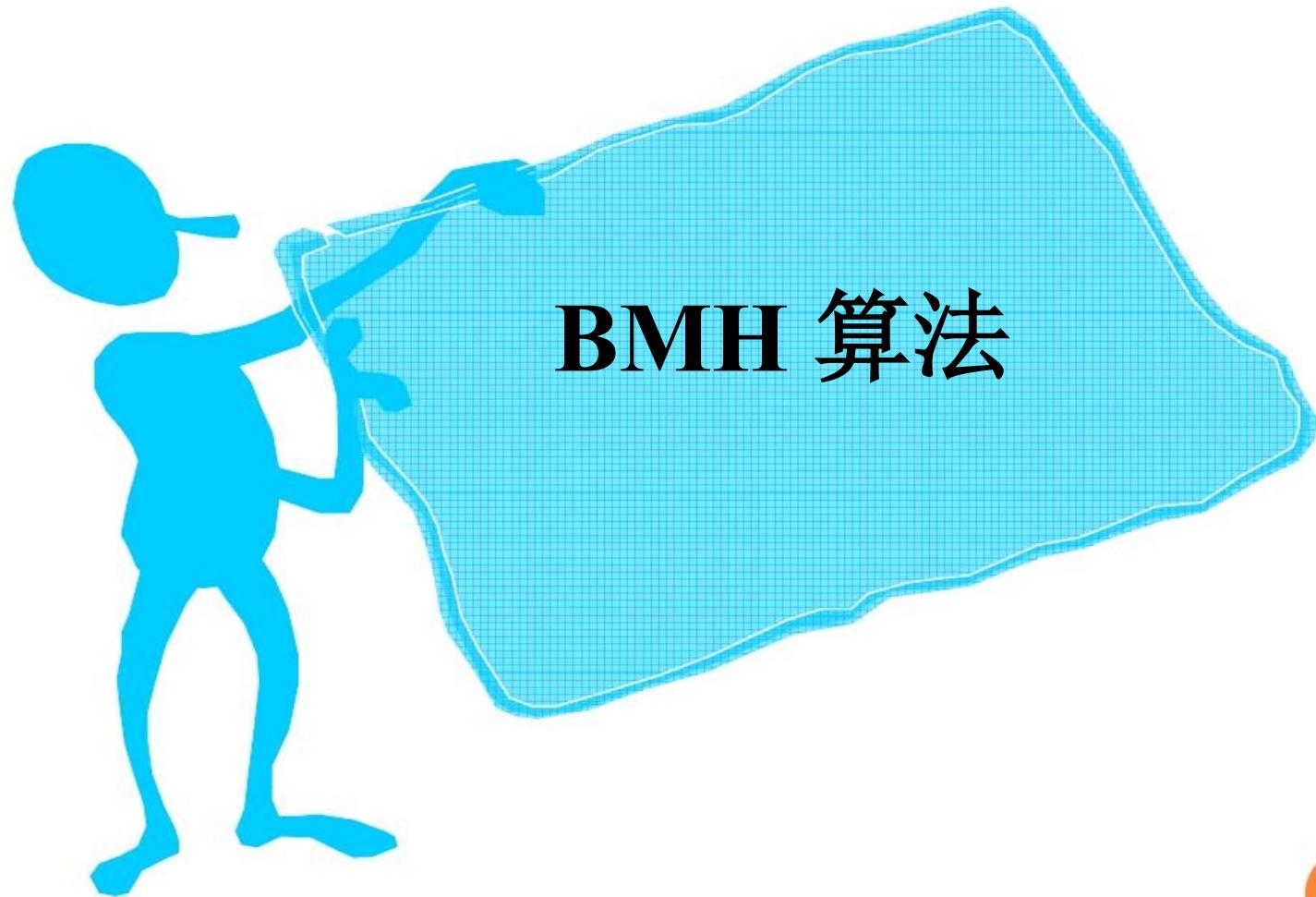


James H. Morris
(1941-)



Vaughan Pratt
(1944-)

BMH 算法



朴素逆向匹配算法

字符串 S ：“BBC ABCDAB ABCDABCDABDE”，模式串 P ：“ABCDABD”

朴素匹配算法

BBC ABCDAB ABCDABCDABDE
ABCDABD



如果从 P 的最右面，向左匹配呢？

朴素逆向匹配算法

BBC ABCDAB ABCDABCDABDE
ABCDABD



朴素逆向匹配算法

Reverse-Naive-Search(T,P)

```
1. For s = 0 To n-m
2.   j←m-1, flag ← 1 // start from the end
3.   // check if  $T[s..s + m - 1] = P[0..m - 1]$ 
4.   While  $T[s+j] = P[j]$  Do
5.     j ← j-1
6.   If j<0 Then
7.     Print “pattern occurs with shift” s
8.     flag ← 0
9.   If flag Then
10.    Return -1
```

- 运行时间和简单算法相同 $O(mn)$

改进朴素逆向匹配算法

- Boyer和Moore在朴素逆向匹配算法中增加了复杂的启发式规则，得到了 $O(n + m)$ 算法，该算法称为Boyer-Moore (BM) 算法
- Horspool建议仅使用简单易实现的出现启发式规则，提出了 Boyer-Moore-Horspool 算法
 - 出现启发式规则：在不匹配发生时，将 $T[s + m - 1]$ (即主串 T 中被比对的最后一个字符) 对齐到模式 $P[0 \dots m-2]$ 中的最右匹配的位置，如果没有匹配的字符，将 P 向右移动 m 个位置

出现启发式规则：在不匹配发生时，将 $T[s + m - 1]$ 对齐到模式 $P[0, \dots, m-2]$ 中的最右匹配的位置，如果没有匹配的字符，将 P 向右移动 m 个位置

例子 $T = \text{"detective date"}$
 $P = \text{"date"}$

T	d	e	t	e	c	t	i	v	e		d	a	t	e
P	d	a	t	e										

T	d	e	t	e	c	t	i	v	e		d	a	t	e
P					d	a	t	e						

T	d	e	t	e	c	t	i	v	e		d	a	t	e
P								d	a	t	e			

T	d	e	t	e	c	t	i	v	e		d	a	t	e
P									d	a	t	e		

为什么出现启发式规则是正确的？

不匹配发生之后， P 必然往右移动，下一个可能的完全匹配，如果包含了 $T[s + m - 1]$ ，则 $T[s + m - 1]$ 必与模式 $P[0 \dots m - 2]$ 中的一个位置匹配，而 P 往右最可靠的移动距离，就是将 $T[s + m - 1]$ 对齐到模式 $P[0 \dots m - 2]$ 中的最右匹配的位置。

出现启发式规则：在不匹配发生时，将 $T[s + m - 1]$ 对齐到模式 $P[0 \dots m - 2]$ 中的 **最右匹配** 的位置，如果没有匹配的字符，将 P 向右移动 m 个位置

例子 $T = \text{“tea kettle”}$
 $P = \text{“kettle”}$

$$T = \text{“}tea kettle\text{”}$$
$$P = \text{“}kettle\text{”}$$

<i>T</i>	t	e	a		k	e	t	t	l	e			
<i>P</i>	k	e	t	t	l	e							

T	t	e	a		k	e	t	t	l	e			
P					k	e	t	t	l	e			

偏移表

- 在预处理中, 计算大小为 $|\Sigma|$ 的偏移表

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

- 例: $P = "kettle"$ (下标从0开始)
 - $shift[e] = 6 - 1 - 1 = 4, shift[l] = 6 - 1 - 4 = 1$
 - $shift[t] = 6 - 1 - 3 = 2, shift[k] = 6 - 1 - 0 = 5$
- 例: $P = "pappar"$, 其偏移表是什么?
 - $shift[p] = 6 - 1 - 3 = 2, shift[a] = 6 - 1 - 4 = 1,$
 $shift[r] = 6$

Boyer-Moore-Horspool 算法

BMH-Search(T,P)

```
1. flag ← 1
2.      // compute the shift table for P
3. For c = 0 To |S|
4.     shift[c] = m
5. For k = 0 To m-2
6.     shift[P[k]] = m-1-k
7. // search
8. s ← 0
9. While s ≤ n-m Do
10.    j←m-1 /*从模式串P的后面往前匹配*/ start from the end
11.    // check if T[s..s + m- 1] = P[0..m- 1]
12.    While T[s+j] == P[j] Do
13.        j ← j-1
14.    If j<0 Then
15.        Print “pattern occurs with shift” s
16.        flag ← 0
17.    s ← s + shift[T[s + m-1]] // shift by last letter
18.    If flag Then
19.        return -1
```

} 计算偏移表

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

字符串查找数据结构

一些字符串匹配的相关问题

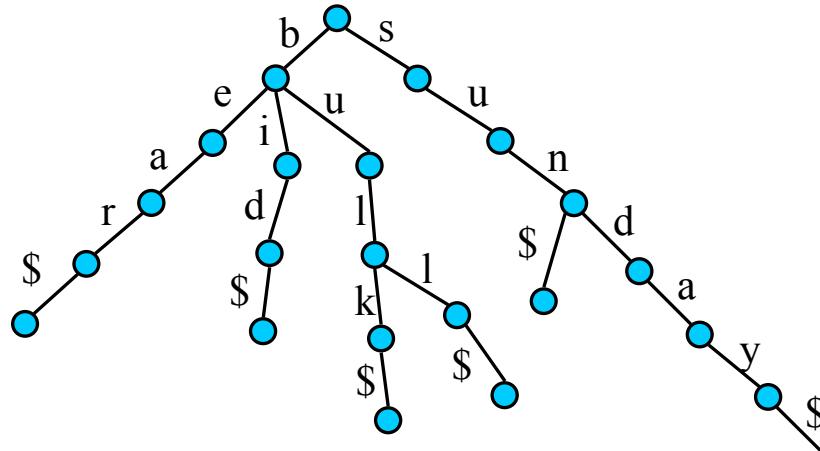
- 给出 N 个单词组成的熟词表，以及一篇全用小写英文书写的文章，请你按最早出现的顺序写出所有不在熟词表中的生词。
- 1000万字符串，其中有些是重复的，需要把重复的全部去掉，保留没有重复的字符串。
- 一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前10个词，请给出思想，给出时间复杂度分析。

字符串查找的抽象数据结构 (Abstract Data Type, ADT)

- 字符串的ADT 存储字符串集合:
 - $\text{search}(x)$ – 查找集合中的字符串 x
 - $\text{insert}(x)$ – 向集合中插入新的字符串 x
 - $\text{delete}(x)$ – 从集合中删除等于 x 的字符串
- 字符串的一些特点
 - 字符串是变长的
 - 很多字符串前缀相同– 可以节约空间

Trie树

- Trie 树，又称字典树，单词查找树或者前缀树，是一种用于快速检索的多叉树结构，如英文字母的字典树是一个26叉树，数字的字典树是一个10叉树。Trie一词来自retrieve，发音为/tri:/ “tree”，也有人读为/traɪ/ “try”。
- Trie树可以利用字符串的公共前缀来节约存储空间，假设每个字符串以“\$”(不在字符表中)结束
- 如果系统中存在大量字符串且这些字符串基本没有公共前缀，则相应的trie树将非常消耗内存，这也是trie树的一个缺点。



字符串集合: {**bear, bid, bulk, bull, sun, sunday**}

Trie树

- trie树的性质：

- 多路树，每个结点有从1到 d 个儿子。
- 根节点不包含字符，除根节点以外每个节点只包含一个字符
- 每个节点的所有子节点包含的字符串不相同
- 每个叶子结点存储字符串，这个字符串是从根到叶子所有字符的连接

Trie的搜索和插入

- 搜索：沿着树向下 (从Trie-Search($root, P[0..m]$)搜索)

Trie-Search($t, P[k..m]$) //search string P from t

1. if (t is leaf) and ($k > m$) then return true
2. else if $t.\text{child}(P[k]) = \text{nil}$ then return false
3. else return Trie-Search($t.\text{child}(P[k]), P[k+1..m]$)

- 插入

Trie-Insert($t, P[k..m]$)

1. if t is not leaf then //otherwise P is already present
2. if $t.\text{child}(P[k]) = \text{nil}$ then
3. Create a new child of t and a “branch” starting
 with that child and storing $P[k..m]$
4. else Trie-Insert($t.\text{child}(P[k]), P[k+1..m]$)

- 删除应该怎么做？

Trie 实现细节

- $t.\text{child}(c)$ 操作的复杂性是什么：
 - 大小为 d 的儿子指针数组：浪费空间，但是 $\text{child}(c)$ 是 $O(1)$
 - 儿子指针的 **hash 表**，较少浪费空间， $\text{child}(c)$ 的期望是 $O(1)$
 - 儿子指针 **链表**：空间小但是 $\text{child}(c)$ 在最坏情况下是 $O(d)$
 - 儿子指针的二分搜索树：空间小且 $\text{child}(c)$ 最坏情况下是 $O(\lg d)$

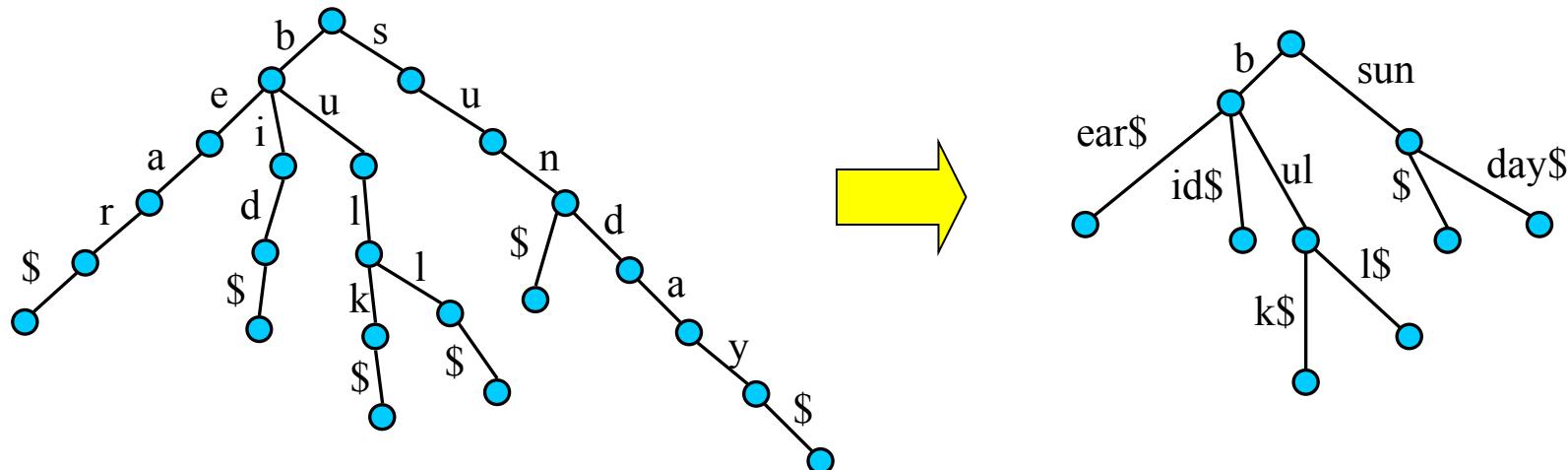
Trie的分析

- 搜索，插入和删除 (字符串长度是 m):
 - 依赖于结点的实现方法，可能为: $O(dm), O(m \lg d), O(m)$
 - Trie的还有很多变种

Trie树的变种

- 紧缩Trie:

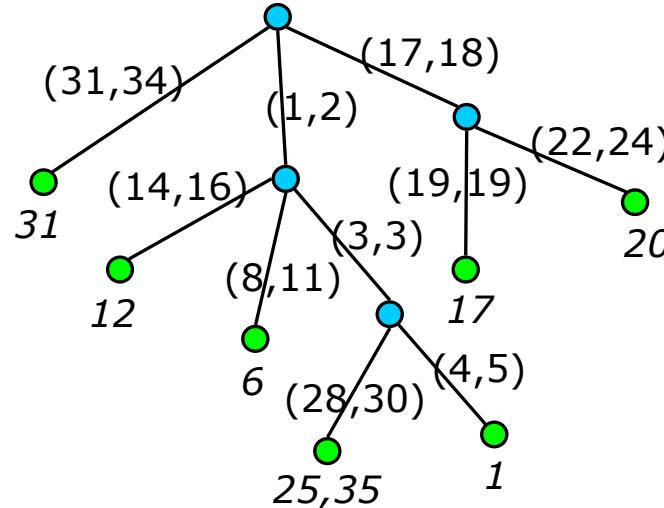
- 用带有字符串的边取代一系列单儿子结点构成的链
- 每个非叶结点最少有两个儿子



紧缩Trie

- 实现:
 - 字符串文本在结构外用一个数组保存，边存放字符在数组中的位置，从而提高Trie树的存储效率
 - 可以用来做单词匹配：找到给定的单词出现在文本的位置。
 - 使用紧缩trie存储文本中所有单词
 - 紧缩trie中的每个儿子保存文档中对应单词出现的位置（单词首字母的位置）。

利用 Trie 进行字符匹配



对于每次边的匹配，都需要去主文本中检索对应的字符串内容

T:

1	2	3	4	5	6	7	8	9	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40
they	th	ink	th	a	t	w	e	w	e	we	were	the	re	and	the	re								

- 查找单词 P :
 - 在每个结点上, 沿着 (i,j) 查找, 从而 $P[i'..j'] = T[i..j]$
 - 如果没有这样的边, T 中没有 P , 否则, 当到达叶子的时候, 找到所有 P 的起始位置

利用 Trie 进行字符匹配

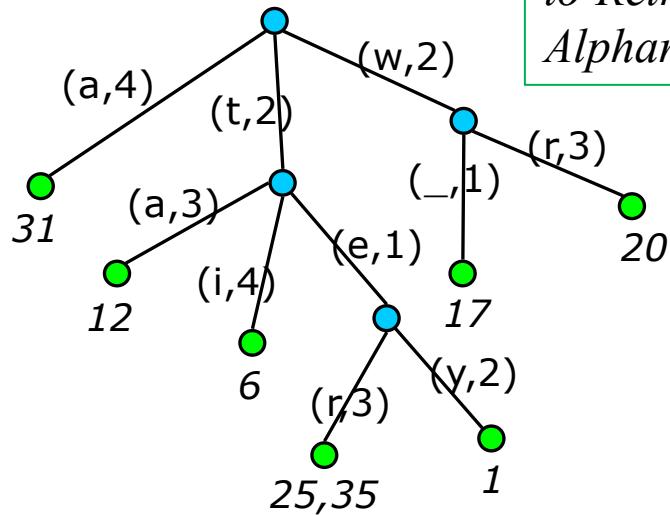
- 根据给定文本建立紧缩:
 - 运行时间: $O(N)$
- 单词匹配的复杂性: $O(m)$

m是搜索key字符串的长度
- 当文本在外存中时?
 - 最坏情况下需要 $O(m)$ 次I/O操作来访问文本中的单个字母-效率不高

Patricia Trie

- *Patricia Trie*:
 - 一种紧缩 trie 其中每个边的标记用 $(T[from], to - from + 1)$ 代替

PATRICIA—Patrical Algorithm
to Retrieve Information Coded in
Alphanumeric



T : **they think that we were there and there**

查询 Patricia Trie

- 单词前缀查询: 查找T中的所有单词, 其前缀是 P[0..m-1]

Patricia-Search(t, P, k) // inserts *P* into *t*

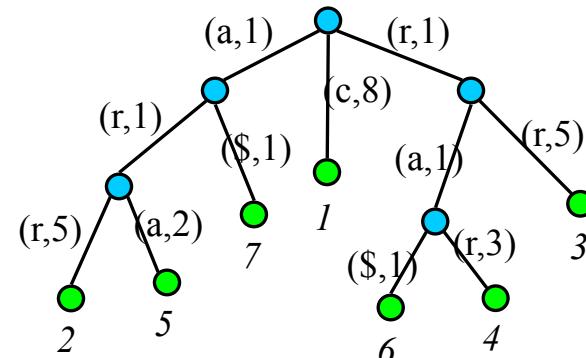
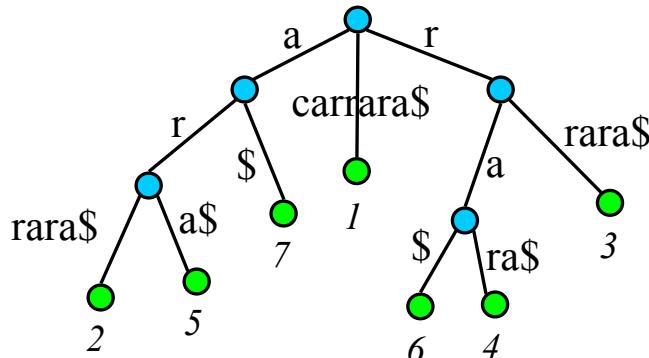
```
01 if t is leaf then
02   j ← the first index in the t.list
03   if T[j..j+m-1] = P[0..m-1] then //整个字符串匹配成功
04     return t.list // exact match
05 else if there is a child-edge (P[k],s) then //t中含有以P[k]为开头, 长度为s的边
06   if k + s < m-1 then //还未扫描到P[m-1]
07     return Patricia-Search(t.child(P[k]), P, k+s)
08   else go to any descendent leaf of t and do the check of line 03
09     if it is true, return lists of all descendent leafs of t, //都是以P为前缀, 且长
10       otherwise return nil                                度大于m的单词
11 else return nil // nothing is found
```

Patricia Trie的分析

- Patricia Trie的想法—将文本的比较推迟到最后
 - 如果文本在外存，仅需要 $O(1)$ 次I/O (如果Trie在内存)

Trie树的变种

- 后缀树：一种包含文本所有后缀的紧缩trie（或类似的结构）
 - 后缀的Patricia trie 有时叫做 Pat 树



1 2 3 4 5 6 7 8
c a r r a r a \$

后缀树应用

- 字符串查找，查找P在主串T中出现位置
- 计算指定字符串 Pattern 在字符串 Text 中的出现次数

算法课没有结束.....