算法设计与分析

第九章 字符串匹配算法

户保田
e-mail:hubaotian@hit.edu.cn

哈尔滨工业大学（深圳）计算机学院

字符串匹配问题

- 字符串 T = “at the **thought** of”
- 字符串 P = “**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 = “at the thought of”
- 字符串 P = “think”

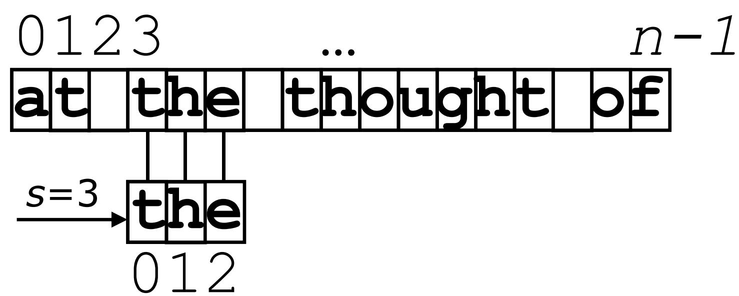
字符串P在T中不存在，所以返回 -1

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:
  - 主字符串 T以及模式串P
- 输出:
  - s – 所有的整数 ($0 \le s \le n - m$) 满足 $T[s .. s+m-1] = P[0 .. m-1]$,

  返回 -1, 如果不存在这样的 s



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”

```
BBC ABCDAB ABCDABCDABDE
ABCDABD
BBC ABCDAB ABCDABCDABDE
 ABCDABD
```

第一位不匹配

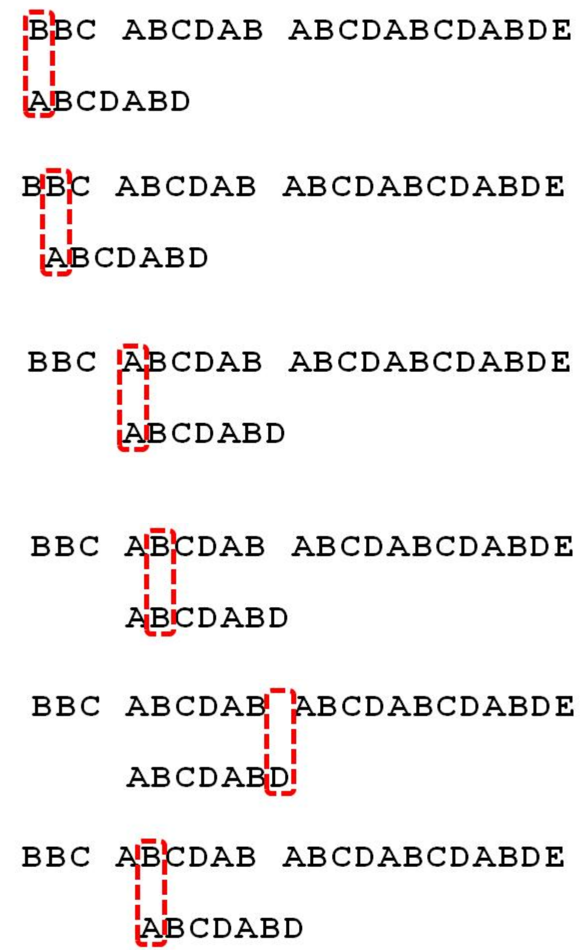
```
BBC ABCDAB ABCDABCDABDE
    ABCDABD

BBC ABCDAB ABCDABCDABDE
    ABCDABD
BBC ABCDAB ABCDABCDABDE
    ABCDABD
```

部分匹配

```
BBC ABCDAB ABCDABCDABDE
     ABCDABD
```

朴素匹配算法

```
Naive-Search(T,P)
flag←1
For s ← 0 to n - m Do
    j ← 1
//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
            print s
            flag←0, s←s+m
If flag Then
    return -1
```



朴素匹配算法的分析

- 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?

  例，T=aaaaaaab, P=aab
- 最好情况: n-m
- 例,T=aaaaaab, P=cde
- 完全随机的文本和模式: O(n–m)

```
Naive-Search(T,P)
flag←1
For s ← 0 to n – m Do
    j ← 1
//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
            print s
            flag←0, s←s+m
If flag Then
    return -1
```

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P的自身组成特点

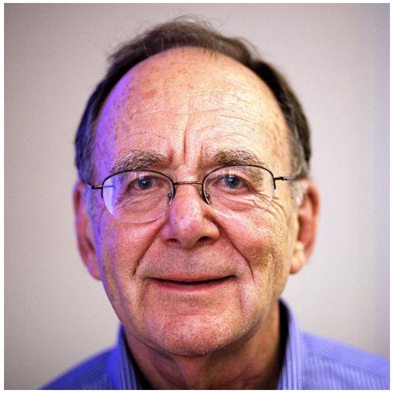
基于指纹的算法

- 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

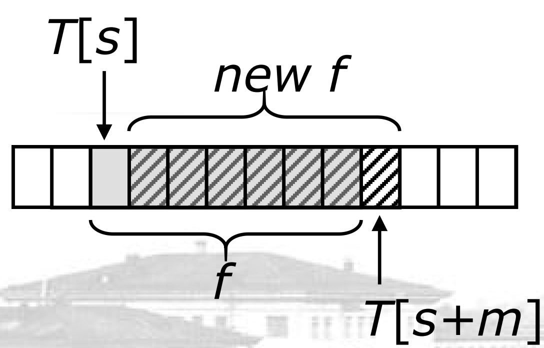
- 令字母表S={0,1,2,3,4,5,6,7,8,9}
- 主字符串T=“921045”，模式串 P=“1045”令指纹为一个十进制数, 即:

$$f(\text{“1045”}) = 1*10^3 + 0*10^2 + 4*10^1 + 5 = 1045$$

基于指纹的算法

主字符串T=“921045”，模式串 P=“1045”

- 可以在O(m)时间计算一个P的指纹 f(P)=1045，f(T[0,..,3])=9210
- 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(T[s+1.. s+m])
  - $f(T[s+1..\,s+m])=(f(T[s\,..\,s+m-1])-T[s]*10^{m-1})*10+T[s+m]$
  - 例，f(T[1,..,4])=(9210-9000)*10+4=2104

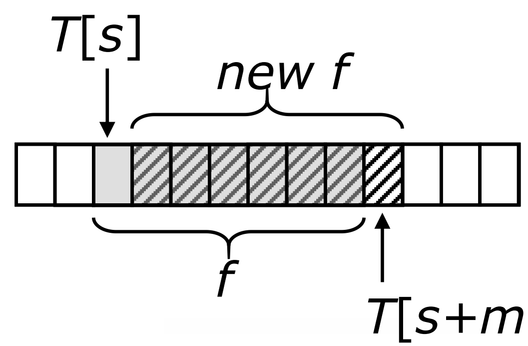


基于指纹的算法

Fingerprint-Search(T,P)

同学们自己动手用伪代码完成上述思想

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



时间复杂度： $2O(m) + O(n-m) = O(n)$!

问题

- 当模式p过长时，即m过大，对应的数值p过大，会导致溢出
- 当两个过大的数值比较大小时，CPU需要多个运算周期来进行，这样两数比较，我们不能假设可以在O(1)时间内完成

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 mod 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，令$q = 7$, h(“73”) = 3, 但 “73” ≠ “52”

这时需要把T[s, ... , s+m-1] 和 P[0...m-1] 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,...,s+m-1] = P[0...m-1]

“mod q”算术运算

公式1:

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

公式2:

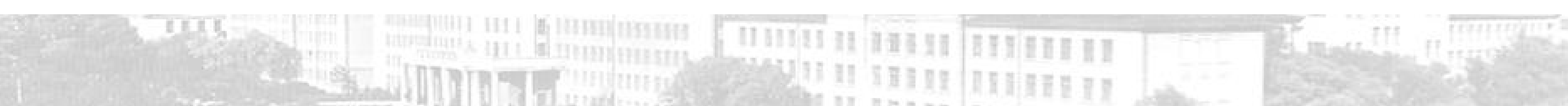
$$(a*b) \bmod q = ((a \bmod q)*(b \bmod q)) \bmod q$$

预处理与步骤

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

基于公式f(T[s+1.. s+m])=(f(T[s .. 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←(ft–T[s]*10^{m-1} mod q)*10+T[s+m])mod q
  - 一般取q大于任意的T[i]的素数，当然q大于10
  - 10^{m-1} mod q 在预处理中计算一次



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,flag← 1
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] then
            print s
              flag← 0
    ft ←((ft – T[s]*c)*10 + T[s+m]) mod q
 if flag then
     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: $\frac{n-m}{q} \times m = O(n-m)$
  - 总时间: O(n-m)

```
Rabin-Karp-Search(T,P)
01 q ← a //prime larger than m
02 c ← 10^(m-1) mod q  // run a loop multiplying by 10 mod q
03 fp ← 0, ft ← 0,flag← 1
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] 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
- Rabin-Karp实现简单，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

朴素匹配算法回顾

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

```
BBC ABCDAB ABCDABCDABDE
ABCDABD

BBC ABCDAB ABCDABCDABDE
 ABCDABD

BBC ABCDAB ABCDABCDABDE
    ABCDABD

BBC ABCDAB ABCDABCDABDE
    ABCDABD

BBC ABCDAB ABCDABCDABDE
    ABCDABD

BBC ABCDAB ABCDABCDABDE
     ABCDABD
```

部分匹配

- 当D与空格不匹配时，我们其实已经知道前面已匹配过的6个字符是"ABCDAB"
- 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?

考虑以下过程:

换个角度

字符串：T=” abababacaba”
模式串：P=” ababaca”

- 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=a, S_1=P[0] = “a”, K_1 = 1　　$K_i=\sigma(S_i)=\max\{k \mid P[0,...,k-1]$是X的后缀$\}$

第2步：X=ab, S_2=P[0,1]= “ab” , K_2 = 2

第3步：X=aba, S_3=P[0,1,2]= “aba”, K_3 = 3

第4步：X=abab, S_4=P[0,1,2,3]= “abab”, K_4= 4

第5步：X=ababa, S_5=P[0,1,2,3,4]= “ababa”, K_5 = 5

第6步：X=ababab, S_6=P[0,1,2,3]= “abab”, K_6 = 4

第7步：X=abababa, S_7=P[0,1,2,3,4]= “ababa”, K_7 = 5

第8步：X=abababac, S_8=P[0,1,2,3,4,5]= “ababac”, K_8 = 6

第9步：X=abababaca, S_9=P[0,1,2,3,4,5,6]= “ababaca”, K_9 = 7

第10步：X=abababacab, S_{10}=P[0,1] = “ab”, K_{10} =2

第11步：X=abababacaba, S_{11}=P[0,1,2] = “aba”, K_{11} = 3

换个角度

字符串：T=" ababababacaba"
模式串：P=" abababaca"

第1步：X=a, S_1=P[0] = "a", K_1 = 1
第2步：X=ab, S_2=P[0,1]= "ab" , K_2 = 2
第3步：X=aba, S_3=P[0,1,2]= "aba", K_3 = 3
第4步：X=abab, S_4=P[0,1,2,3]= "abab", K_4= 4
第5步：X=ababa, S_5=P[0,1,2,3,4]= "ababa", K_5 = 5
第6步：X=ababab, S_6=P[0,1,2,3]= "abab", K_6 = 4
第7步：X=abababa, S_7=P[0,1,2,3,4]= "ababa", K_7 = 5
第8步：X=abababac, S_8=P[0,1,2,3,4,5]= "ababac", K_8 = 6
第9步：X=abababaca, S_9=P[0,1,2,3,4,5,6]= "ababaca", K_9 = 7
第10步：X=abababacab, S_{10}=P[0,1] = "ab", K_{10} =2
第11步：X=abababacaba, S_{11}=P[0,1,2] = "aba", K_{11} = 3

分析:

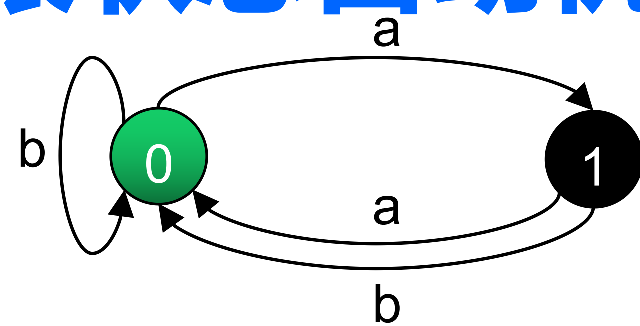
1、需将T的字符都读入，O(n)

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，$M=\{Q，q_0，A，\sum，\delta\}$:

- 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。
- $\sum$所有可能的输入字母表集合。
- δ被称为状态转移函数，$Q \times \sum -> Q$的一个映射函数

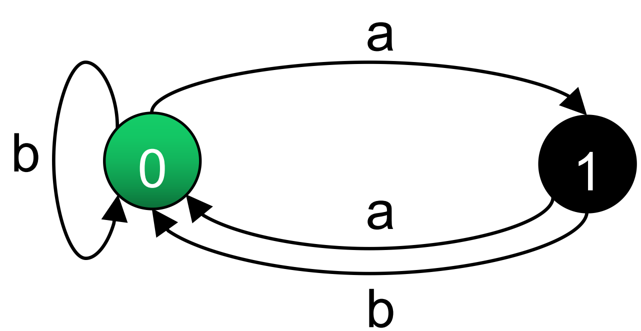
有限状态自动机

状态集合：$Q=\{0,1\}$

初始状态：$q_0=0$

接受状态：$A=\{1\}$

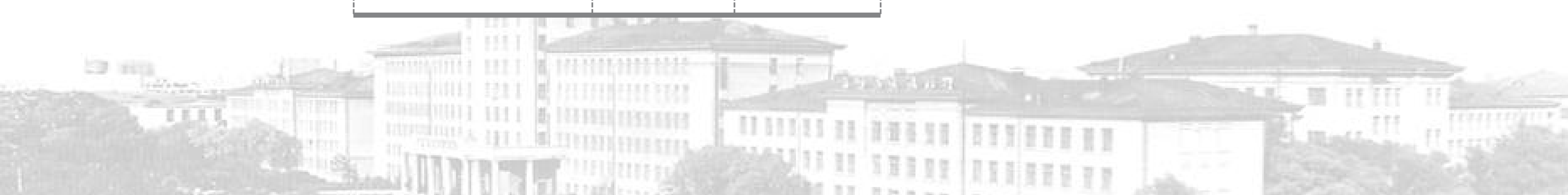
字母集合：$\Sigma=\{a,b\}$



状态转移函数$\delta=$

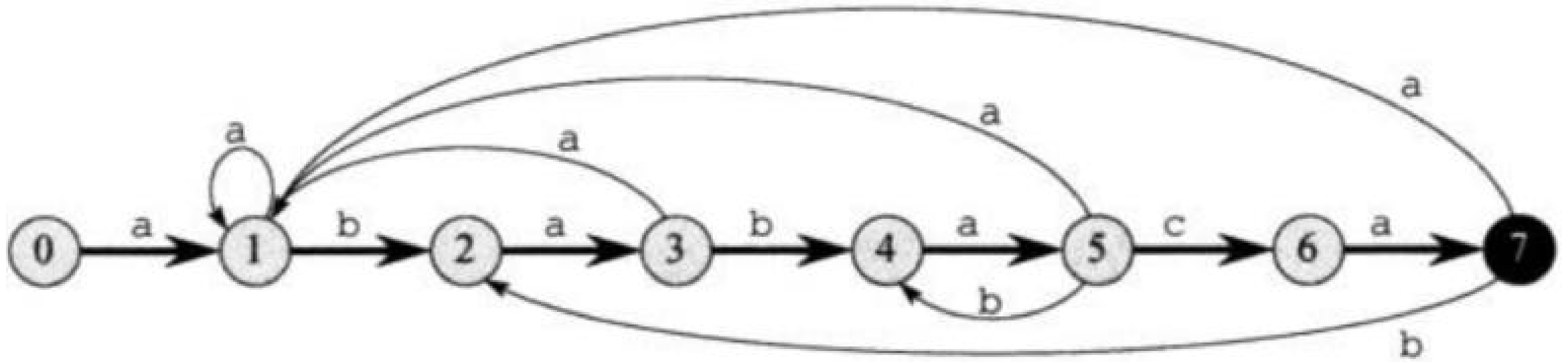
状态 \ 输入	**a**	**b**
0	1	0
1	0	0

- 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}，P与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}

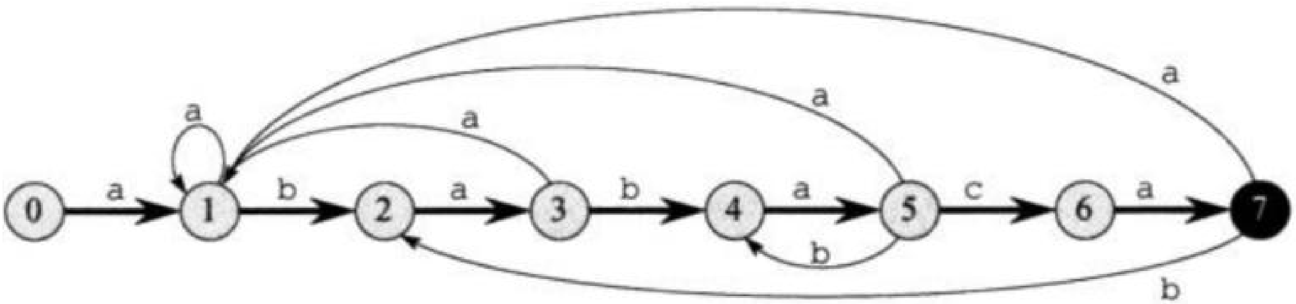
字母集合：Σ={'a','b','c'}

转移函数：$\delta(q, \alpha) = \sigma(P[0, \ldots, q-1]\alpha)$

=max{k | P[0, . . . , k−1]是P[0, . . . , q−1]α的后缀}

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”



- δ（q, α）= σ（P[0, q]α）

$\delta =$

=max{k|P[0, . . . , k−1]是P[0, . . . , q−1]α的后缀}

- 状态转移函数可表示为一个（m+1）|Σ|的表

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

有限状态自动机字符串匹配算法

```
Finite-Automation-Matcher(T,P,Σ)
n←T.length
q←0,flag←1
δ=←Commpute-Transition-Table(P,Σ)
for i←0 to n
        q←δ(q,T[i])
        if q=m then
              Print "pattern occurs with shift" i-m+1
              flag←0
if flag then
    return -1
```

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

构造状态转移表

```
Commpute-Transition-Table(P,Σ)
m←P.length
for q←0 to m
    for each character a in Σ
        k←min(m+1,q+2)
        repeat
            k←k-1
        until P[0...k-1]is the suffix of P[0...q-1]a
        δ(q,'a')←k
return δ
```

最坏时间复杂度O(m³|Σ|)，可以改进为O(m|Σ|)（课后练习**32.4-8**）

Knuth-Morris-Pratt 算法

- 避免预先计算（$m+1$）$|\Sigma|$的状态转移表$\delta(q, \alpha)$
- 忽略尚未进行匹配的字符α，只利用已匹配字符$P[0..q-1]$的信息
- 从而使得预先需计算2维（$m+1$）$|\Sigma|$的表简化为1维数组$\pi(q)$

前缀表π

- π [q] = max{k < q | P[0..k-1] = P[q−k..q-1]}
- q：模式串P与主字符串中已匹配上的字符个数
- 给定模式串P=“ABCDABD”，计算π [5]:
  - P[0..4]='ABCDA'
  - 前缀包括，A,AB,ABC,ABCD
  - 后缀包括，A,DA,CDA,BCDA

	0	1	2	3	4	5	6	7
π	0	0	0	0	0	1	2	0

*

预先构造前缀表π

```
Commpute-Prefix(P)
m←P.length
Let π[0,...,m] be a new array
π[0]←0， π[1]←0
for q←2 to m
    k←π[q-1]
    while k>0 and P[k] !=P[q-1]
        k=π[k]
    if P[k] =P[q-1]
        k←k+1
    π[q]←k
return π
```

q：模式串P与主字符串中已匹配上的字符个数

p: 模式串的下标从0开始

用摊还分析来做

时间复杂度：Θ(m)

Knuth-Morris-Pratt 算法

```
KMP-Search(T,P)
m←P.length
n←T.length,flag←1
π ← Compute-Prefix(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]//next character dons not match
        q ← π[q]
    if P[q] = T[i]//next character matches
        q ← q + 1
    if q = m //all P's characters matched
        Print "pattern occurs with shift" i-m+1, flag←0
        q ← π[q]
  if flag then
   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

往前移动位数 = q - $\pi[q]$

已匹配的字符数

对应前缀表的值

```
BBC ABCDAB ABCDABCDABDE
    ABCDABD
```

部分匹配

```
BBC ABCDAB ABCDABCDABDE
    ABCDABD
```

朴素匹配算法

```
BBC ABCDAB ABCDABCDABDE
        ABCDABD
```

KMP算法

部分匹配q=6，可能有用π[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)
m=P.length
n=T.length
π ← Compute-Prefix(P)
q ← 0
for i ← 0 to n-1
    while q > 0 and P[q] ≠ T[i]
        q ← π[q]
    if P[q] == T[i]
        q ← q + 1
    if q == m
        Print  i-m+1
    q ← π[q]
```

```
Finite-Automation-Matcher(T,P,Σ)
n=T.length
q=0
δ=Commpute-Transition-Table(P,Σ)
for i=0 to n-1
        q=δ(q,T[i])
        if q==m
            Print i-m+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-）

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

同学们动手写一下朴素逆向匹配算法

```
Reverse-Naive-Search(T,P)
for s ← 0 to n − m
    j ← m − 1, flag←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
        print “pattern occurs with shift” s
        flag←0
if flag then
  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向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

  就是加一些人的经验

  - 出现启发式规则：在不匹配发生之后，将T[s + m–1]对齐到模式 P[0..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

示例1

出现启发式规则：在不匹配发生之后，将T[s + m−1]对齐到模式P[0..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

T= "detective date"　P= "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

示例2

出现启发式规则：在不匹配发生之后，将T[s + m−1]对齐到模式P[0..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

T=“tea kettle” P=“kettle”

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

T	t	e	a		k	e	t	t	l	e				
P					k	e	t	t	l	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，如包含了T[s + m−1]，则T[s + m−1]必与模式 P[0..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..m−2]中的最右匹配的字符

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`**] =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)
flag←1
  // compute the shift table for P
for c ← 0 to |Σ|
   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 then
      print "pattern occurs with shift" s
      flag←0
   s ← s + shift[T[s + m–1]]  // shift by last letter
if flag then
  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

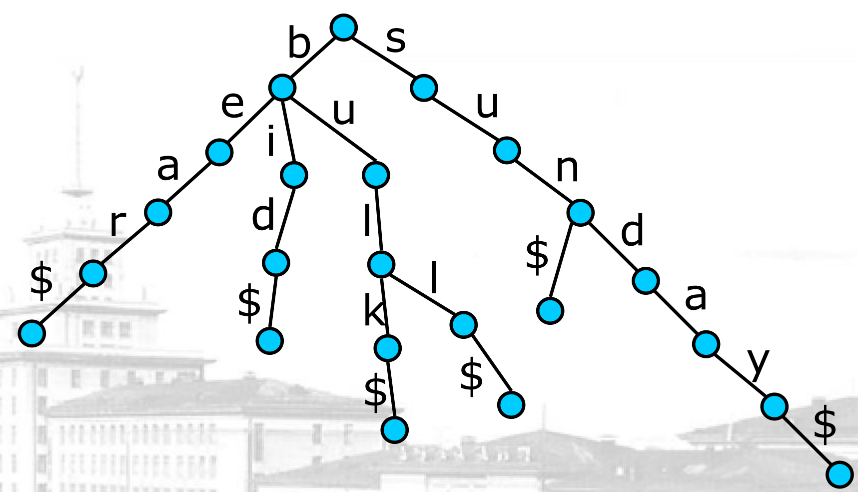
字符串查找数据结构（选修）

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:
  - search(x) – 查找集合中的字符串x
  - insert(x) – 向集合中插入新的字符串x
  - 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，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d 个儿子节点
  - 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路径上的所有字符

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
if t is leaf then return true
else if t.child(P[k])=nil then return false
     else return Trie-Search(t.child(P[k]), P[k+1..m])
```

插入

```
Trie-Insert(t, P[k..m])
if t is not leaf then//otherwise P is already present
    if t.child(P[k])=nil then
     Create a new child of t and a "branch" starting with
     that chlid and storing P[k..m]
    else Trie-Insert(t.child(P[k]), P[k+1..m])
```

Trie 实现细节

- t.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child(c) 时间复杂度：O(1)
  - 儿子指针的hash表，较少浪费空间，child(c) 的期望时间复杂度： O(1)
  - 儿子指针链表：空间小但是child(c) 最坏情况下时间复杂度：O(d)
  - 儿子指针的二分搜索树: 空间小，child(c)最坏情况下时间复杂度：O(lgd)

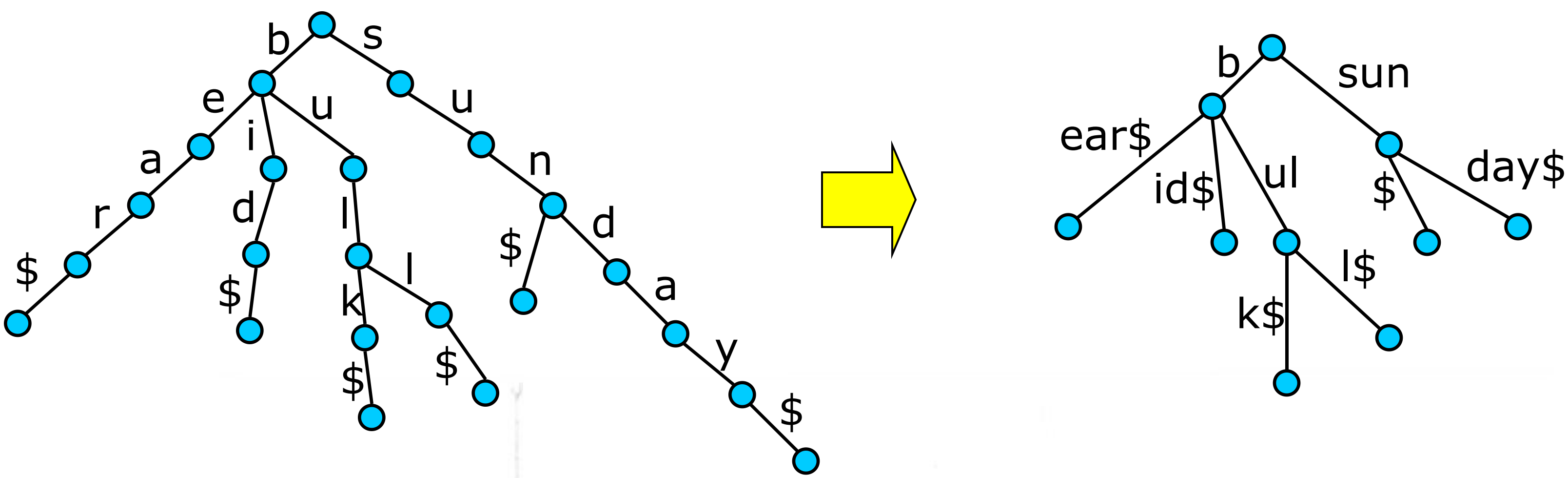
Trie的分析

- 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

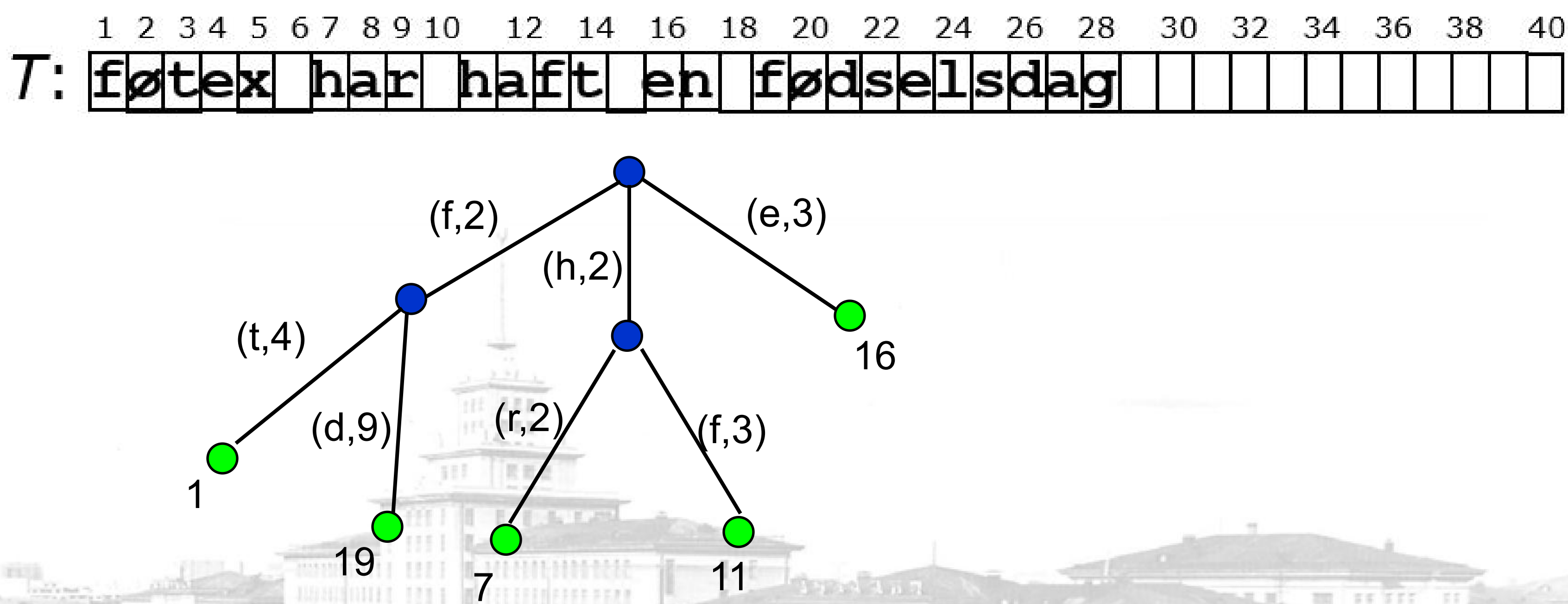
Trie树的变种

- 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



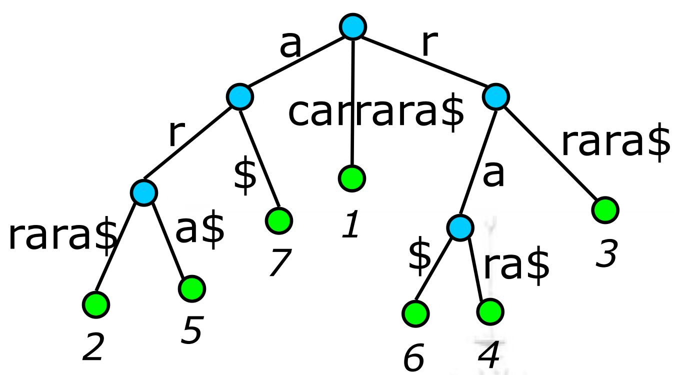
Trie树的变种

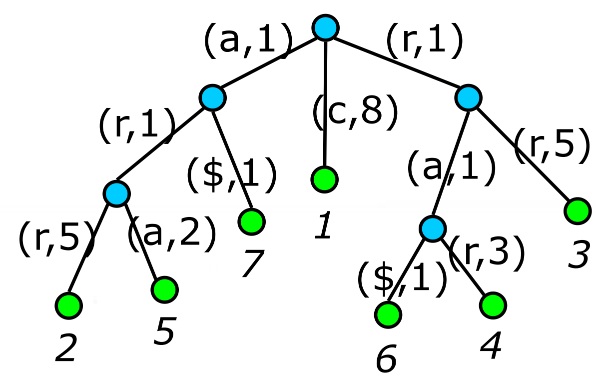
- Patricia trie:
  - PATRICIA—Patrical Algorithm to Retrieve Information Coded in Alphanumeric,它一种紧缩 trie 其中每个边的标记 用 (T[from], to – from + 1）代替



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	$

算法课没有结束……