字符串 String

1 基本信息

1.1 介绍

字符串，特殊的 线性表，即元素为 **char** 的线性表

n (≥ 0) 个字符的有限序列，n ≥ 1时，一般记作 S : “c0c1c2...cn-1”

- S 是串名
- “c0c1c2...cn-1”是串值
- ci是串中的字符
- N 是串长（串的长度）： 一个字符串所包含的字符个数
- **空串**：长度为**0**的串，它不包含任何字符内容（注意 与 空格串“ ”的区别）

1.2 特殊线性结构

	基本线性结构	**字符串**
数据对象	无特殊限制	串的数据对象为字符集
基本操作	线性表的大多以“单个元素”为操作对象	串通常以“串的整体（子串）”作为操作对象
存储方式	数组、链表等	数组、链表等

1.3 子串

子串的定义

$$
\begin{aligned}
&\text{假设} s1，s2\text{是两个串：}\\
&s1 = a_0a_1a_2\dots a_{n-1}\\
&s2 = b_0b_1b_2\dots b_{m-1}\\
&\text{其中} 0 \le m \le n, \text{若存在整数} i(0 \le i \le n-m)\\
&\text{使得} b_j = a_{i+j}, j = 0,1,\dots,m-1 \text{同时成立}\\
&\text{则称串} s_2 \text{是串} s_1 \text{的子串，} s_1 \text{为串} s_2 \text{的主串，或称} s_1 \text{包含串} s_2
\end{aligned}
$$

特殊子串

- **空串**是任意串的子串
- 任意串 S 都是 **S 本身** 的子串
- 真子串：非空且不为自身的子串

1.4 字符串的基本操作

假设

s1= "SUSTech"

s2 = "10"

操作	解释	示例
拼接append	向字符串后面添加	s1.append(s2) s1 = "SUSTech10"
赋值assign	把内容赋值给字符串	s1.assign(s2) s1 = "10"
插入insert	插入到字符串的某个位置	s1.insert(s2,0) s1 = "10SUSTech"
删除erase	清空字符串的内容	s1.erase() s1 = ""
替换replace	替换子符串某一特定位置的值	s1.replace('s',0) s1 = "sUSTech"
交换swap	交换字符串的值	s1.swap(s2) s1 = "10", s2 = "SUSTech"
查找find	找到字符串里面某个特定的字符	s1.find(0) char_find = 'S'

❓问题：求子串数量

> Give string s="SUSTechCS203", how many sub string it has?

$$s.len = 12$$

因为包括注意空字符串

$$substring = \frac{(1+s.len)(s.len)}{2} + 1(\text{空字符串}) = 79$$

❓问题：S1 + S2 == S2 + S1 成立的所有可能条件

> Set S1 and S2 as strings, please give the possible conditions for S1+S2 == S2+S1 to be true (where + is the join operation)

$$1.S1\text{和}S2\text{均为空串}$$
$$2.S1\text{为空串，}S2\text{为任意字符串}$$
$$3.S2\text{为空串，}S1\text{为任意字符串}$$
$$4.S1 == S2\text{两个字符串完全相等}$$
$$5.S1 \neq S2\text{，则长串由整数个短串组成}$$

1.5 字符串的应用

- 文字处理器Word processors
- 病毒扫描Virus scanning

  定义产生病毒的软件都是凭借在软件里查找是否有与病毒软件定义的字符串相似的进行匹配

  如果出现可以的字符就会报警
- 文本检索Text retrieval
- 自然语言处理Natural language processing

  机器人识别、处理语言
- 网络搜索引擎Web search engine

2 字符串的匹配

2.1 介绍

模式匹配 (pattern matching)

用给定的模式P，在目标字符串T 中搜索与模式P 全同的一个子串，并求出T 中第一个和P 全同匹配的子串（简称为"配串"）,返回其首字符位置

输入

- 一个目标对象 T（字符串）
- search pattern P（字符串）

输出

匹配的首字符位置

2.2 应用

文本编辑时的特定词、句的查找

- 字(词)处理技术
- Web搜索
- 桌面搜索

DNA 信息的提取（计算生物学的应用）

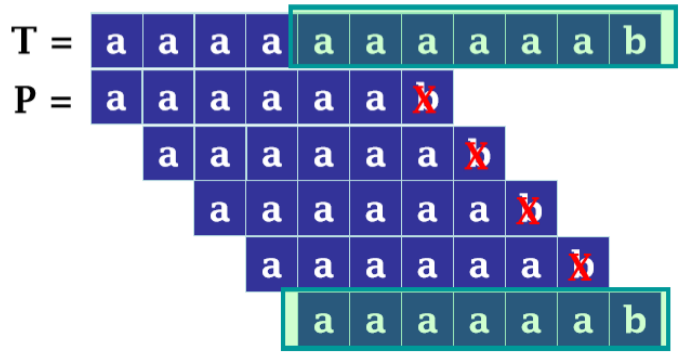
确认是否具有某种结构

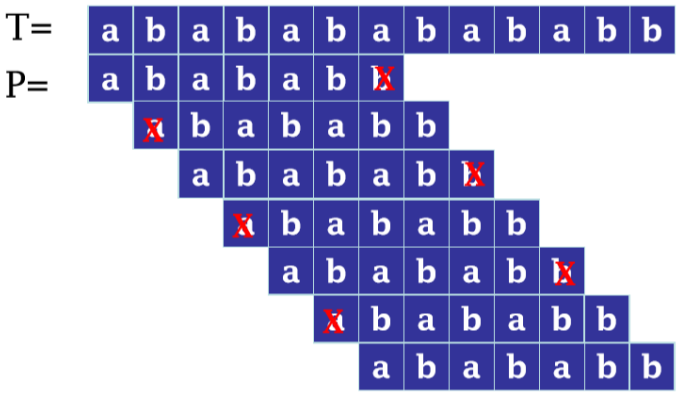
在Unix中使用的Grep操作

2.3 穷举法 Brute Force

思路

逐个字符匹配直到全部匹配完成





伪代码

```
// 一个目标对象 T（字符串）
//search pattern P（字符串）
Algorithm: Brute Force(T, P):
n ← len(T)
m ← len(P)
for i ← 0 to n-m-1 //注意是n-m-1
    for j ← 0 to m-1
        if P[j] != T[i + j] then
            break
    if j = m-1 then
        print('pattern occurs with in' + i)
    end
end
```

时间复杂度分析 O(mn)

在最坏的情况下，每一次循环都不成功，则一共要进行比较（n-m+1）次

每一次"相同匹配"比较所耗费的时间，是 P 和 T 逐个 字符比较的时间，最坏情况下，共 m 次

因此，整个算法的最坏时间开销估计为 O(mn)

2.4 Rabin-Karp算法

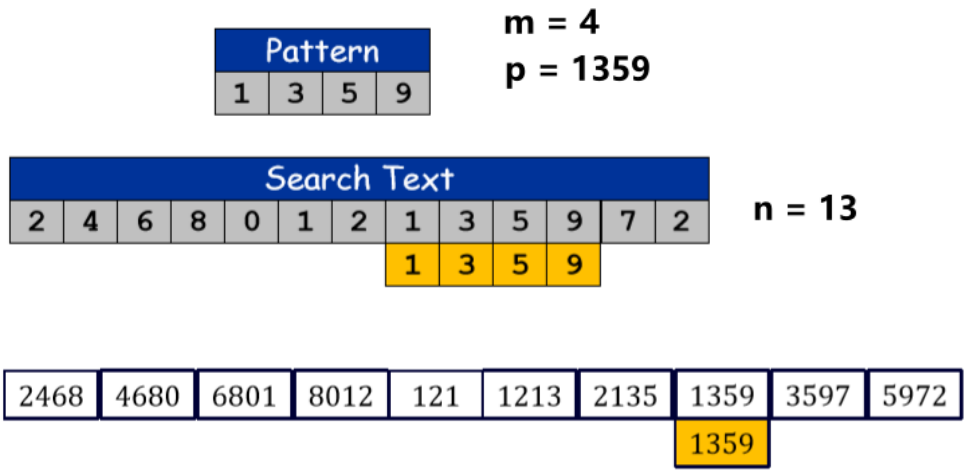
思路：Rabin-Karp

将 P 转换成某一个数字p

将 T 分成 n-m+1 个数组，t[0],t[1],...t[n - m]

将 p 与 t[i] 依次比较

如果 p = t[i] 则 P 出现



思路：将字符转换为数字O(nm)→O(n)

假设条件

- 目标对象：T
- String Pattern: P
- 目标对象的转换数字每个小隔间：t
- String Pattern转换成数字：p
- 目标对象的总长度：n
- String Pattern的总长度：m
- 字母表：Σ
- 字母表大小（全体字符的个数为）d
- 取模的模数q

转换数字（进制转换）O(1)

$$p = P[m-1] + d(P[m-2] + d(P[m-3] + \ldots + d(P[1] + dP[0])))$$

$$p = d^{m-1}P[0] + d^{m-2}P[1] + \ldots + d^{1}P[m-2] + d^{0}P[m-1]$$

当 m 很大的时候，p可能太大了，把p取模

$$p = P[m-1] + dP[m-2] + \ldots + d^{m-1}P[0] \mod q$$

每一个t[i]的转换O(1)（可以用十进制推算）

$$t[0] = d^{n-1}T[0] + d^{n-2}T[1] + \ldots + d^{1}T[m-2] + d^{0}T[m-1]$$

$$t[i+1] = d(t[i] - d^{m-1}T[i]) + T[i+m]$$

如果d^{m-1}太大，t太大，还可以进行取模

$$t[i+1] = d(t[i] - hT[i]) + T[i+m] \mod q$$
$$where\ h = d^{m-1} \mod q$$

时间复杂度分析

t[0] -> t[n - m]

故时间复杂度为**O(n-m)**

伪代码：Rabin-Karp

注意

因为我们取模了！！

即使$p \equiv t[i]\ (mod\ q)$，这也不能说明 $p = t[i]$

还需要进一步的验证

```
Algorithm:Rabin-Karp(T,P,d,q):

n ← len(T)
m ← len(P)
h ← d^{m-1} mod q
p ← 0
t[0] ← 0

//计算初始p和t0
for j ← 0 to m-1
    p ← (d * p + P[j]) mod q
    t[0] ← (d * t[0] + T[j]) mod q
end
//顺序比较
for i ← 0 to n-m
    //不相同，必然有错
    if p != t[i] then
        t[i+1] = (d * (t[i] - h * T[i]) + T[i+m])  mod q
    //相同，还需要进一步对比
    else
        //内部比较出现不同
        for k ← 0 to m-1
            if P[k] != T[i + k] then
                break
        end
        //内部比较完全相同
        if k = m-1 then
            print("pattern occurs with in" + i)
end
```

时间复杂度分析 O(mn)

在最坏的情况下，每一次匹配都成功，则一共要进行比较（n-m+1）次

每一次“相同匹配”比较所耗费的时间，是 P 和 t[i] 逐个 字符比较的时间，最坏情况下，共 m 次

因此，整个算法的最坏时间开销估计为 O(mn)

2.5 有限状态自动机FSA

有限状态自动机的定义

Q：一系列状态集

$q_0 \in Q$：初始状态

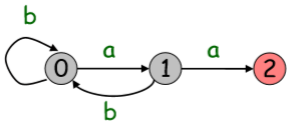
$A \subseteq Q$：结束状态**（注意是集合）**

Σ：输入字母表

δ：状态转移函数（数量为$Q \times \Sigma$）

	0	1
a	1	2
b	0	0

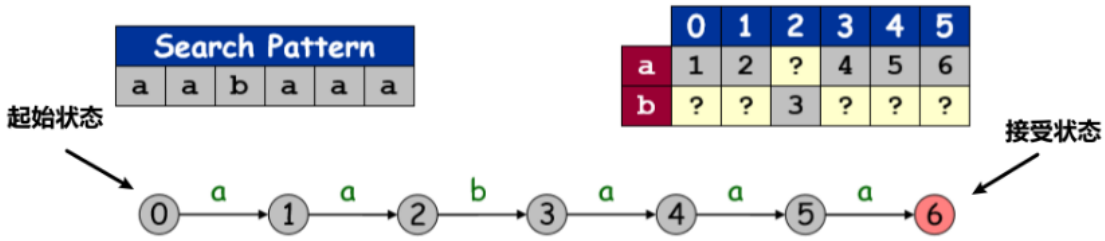
状态转移函数表



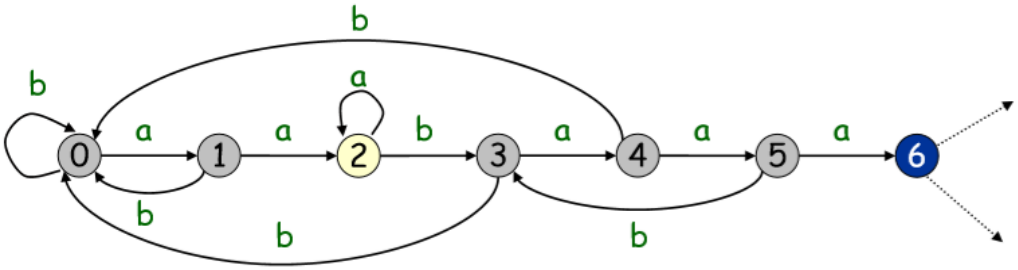
有限状态自动机的图示

思路

核心就是填表



如何填写状态转移表/如何画状态转移函数



问题：绘制状态机

> Build FSA for aabaaabb

$Q = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$

$q_0 = 0$

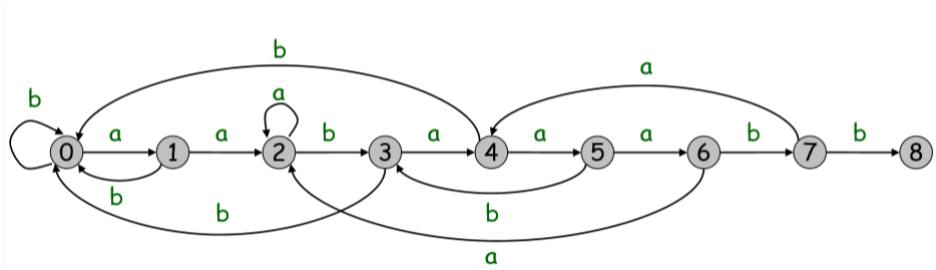
$A = \{8\}$

$\Sigma = \{a, b\}$

状态转移函数δ如下图所示

Search Pattern							
a	a	b	a	a	a	b	b

	0	1	2	3	4	5	6	7
a	1	2	2	4	5	6	2	4
b	0	0	3	0	0	3	7	8



❗伪代码

```
Algorithm:FSA(T,P):
n ← len(T)
m ← len(P)
δ ← Transition(P,Σ)//建立δ的时间复杂度是O(mΣ)，但是通常比较小，所以可以估计为O(m)
q ← 0
for i ← 0 to n-1
    //新的状态 = 当前状态+新遇到的数
    q ← δ(q, T[i])
    if q = m
        print('pattern occurs with in' + (i - m))
end
```

时间复杂度分析 O(n + mΣ)→O(n + m)

建立δ的时间复杂度是O(mΣ)，但是通常比较小，所以可以估计为O(m)

遍历T的时间是O(n)

因为两者是顺序建立的，所以总时间复杂度为O(n + mΣ)→O(n + m)

FSA用于KMP算法

如果匹配，进入**下一个状态**

只需要跟踪字符**不匹配**时去哪里

- 如果状态j中的字符不匹配，进入状态next[j]

2.6 KMP算法

思路

前缀串(P[0,...])

- 字符串S的aprefix是S的子字符串发生在S的开头
- P[0,...,0] = “H” (note that P[0]=‘H’), P[0,...,1] = “He”, P[0,...,4] = “Hello”, 记为: **P[0,...]**

后缀串(P[...,m])

- 字符串S的后缀是发生在S末尾的S的子字符串

- P[9,...,9]="3", P[7,...,9]="203", P[5,...,9] = "CS203", 记为: **P[...,m]**

通常，FSA的构造是为了让状态数告诉我们有多少前缀P被匹配

FSA的转移函数

查找P的最长前缀也是T的后缀[...，i]，表示为k，即，P[1，...，k]=T[i-k，...，i]

（注意不包括P[0]，直接从1开始）

读下一个字母"k+1"，有两种过渡

- P[k+1] = T[i+1]，它匹配，继续
- 否则,它是不匹配的,去δ(k, T [m + 1])

KMP就是相当于算出来δ(k, T [m + 1])，其它都不管

❓问题：填表P[i]和Π[i]

> Example: P="abababaca"

本质上求a[0,...,k]和a[n-k+1...,n-1]有没有匹配

前缀最后一个不能到最后一个a[n-1]

后缀最初一个不能到最前面一个a[0]

I	0	1	2	3	4	5	6
P[i]	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

ababa

a-a

aba-aba 2个

abab

ab

Seq	**0**	**1**	**2**	**3**	**4**	**5**	**6**
P[i]	a	b	a	b	a	c	a
x前	x	a	ab	aba	abab	ababa	ababac
x	a	ab	aba	abab	ababa	ababac	ababaca
Π[i]	0	0	1	2	3	0	1

我们看**x行**的字符串，看它的前缀（不包括最后一个）和后缀（不包括最初一个）可以回到**x前**行的哪一个状态

假如

Seq	0	1	2	3	4	5	6	7	8	9	10
T	a	b	a	b	b	a	b	a	b	a	b
P[i]	a	b	a	b	a	c	a				
Π[i]	0	0	1	2	3	0	1				

在Seq=0的时候，T[0]=P[0]，TP指针均右移

在Seq=1的时候，T[1]=P[1]，TP指针均右移

在Seq=2的时候，T[2]=P[2]，TP指针均右移

在Seq=3的时候，T[3]=P[3]，TP指针均右移

在Seq=4的时候，T[4]≠P[4]

P往回看一个Π[3]=2

在Seq=2的时候，T[4]≠P[2]

P往回看一个Π[1]=0

在Seq=0的时候，T[4]≠P[0]

没有办法了，T指针右移动

在Seq=0的时候，T[5]=P[0]，TP指针均右移

在Seq=1的时候，T[6]=P[1]，TP指针均右移

在Seq=2的时候，T[7]=P[2]，TP指针均右移

在Seq=3的时候，T[8]=P[3]，TP指针均右移

在Seq=4的时候，T[9]=P[4]，TP指针均右移

在Seq=5的时候，T[10]≠P[5]

P往回看一个Π[4]=3

在Seq=3的时候，T[10]=P[3]，TP指针均右移

…

在Seq=n的时候，T[n]=P[6]，输出n-6的值为连续值，T右移，P去Π[6]=1处

在Seq=n+1的时候，比较T[n+1]与P[1]

…

	0	1	2	3	4	5	6	7
T	a	b	c	d	a	b	c	a
x	a	ab	abc	abcd	abcda	abcdab	abcdabc	abcdabca
Π[i]	0	0	0	0	1	2	3	1

Π[i]的意义也在于在**x行**中到此为止的**字符串的最长公共前后缀的大小是多少**

❗伪代码——Π[i]的填写

```
Algorithm:NextArray(P):

m ← len(p)
Let Π[1,…,m] be a new array
Π[1] = 0
k ← 0//初始状态

for j ← 2 to m

    while(k > 0 && P[k+1] != P[j])
        k ← Π[k]
    end

    if P[k+1] == P[j] then
        k ← k + 1

    Π[j] = k

end

return Π[]
```

❗伪代码——KMP实现

```
Algorithm:KMP(T,P):
n ← len(T)
m ← len(P)
q ← 0//初始状态

for i ← 1 to n

    while q > 0 and P[q+1] != T[i]
        q ← Π[q]
    end

    if P[q+1] = T[i] then
        q ← q + 1

    if q = m
        print("pattern occurs with shift" +  (i – m))
        q = Π[q]
```

```
18
19 end
```