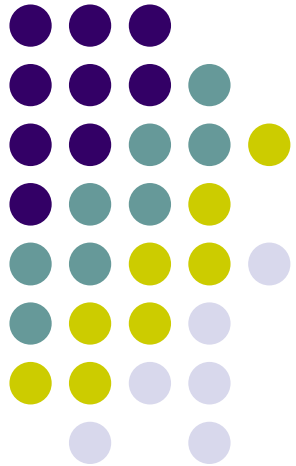
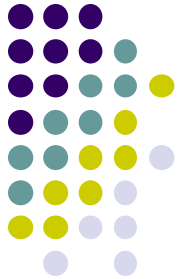
字符串及模式匹配

吉林大学计算机学院
谷方明
fmgu2002@sina.com

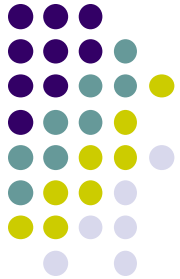


字符串基本定义



- 字符串:一个由字符组成的有限序列，简称串
  - ✓ string 和 char
  - ✓ 字符串可认为是由字符构成的线性表
- 一般把字符串记作： S="$a_0a_1\ldots a_{n-1}$"，S是串名，引号中的字符序列是串值，字符个数n是串长度。
- 空串：长度为零的串称为空串。
  - ✓ 空串与空格字符

字符串术语

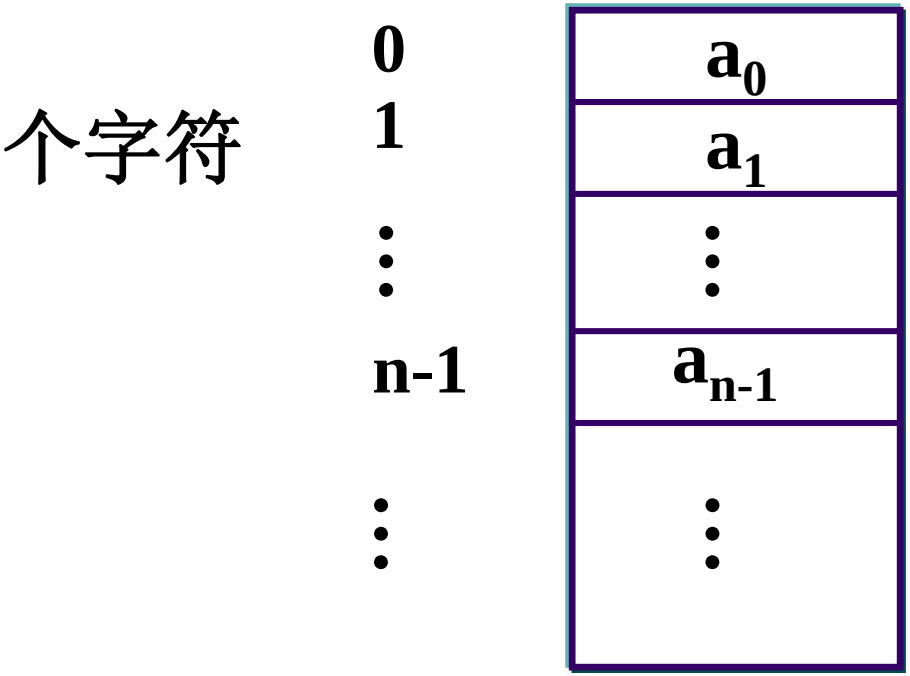


- 子串：串中任意个连续的字符组成的子序列被称为子串。相对于子串该串称为主串。
  - ✓ 规定：空串是任意字符串的子串
- 前缀：若字符串 x = wy, 则称w是x的前缀
- 后缀：若字符串 x = yw, 则称w是x的后缀

- 规定：空串是任何字符串的前缀和后缀
- 前缀后缀关系都具有传递性

字符串的顺序存储

- 顺序存储：把一个串所包含的字符序列相继存入连续的内存中
- 一个存储单元存放一个字符

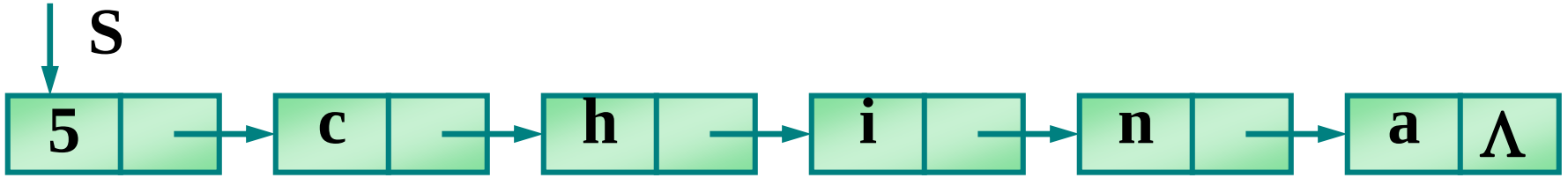


- 压缩存储：一个存储单元存放多个字符

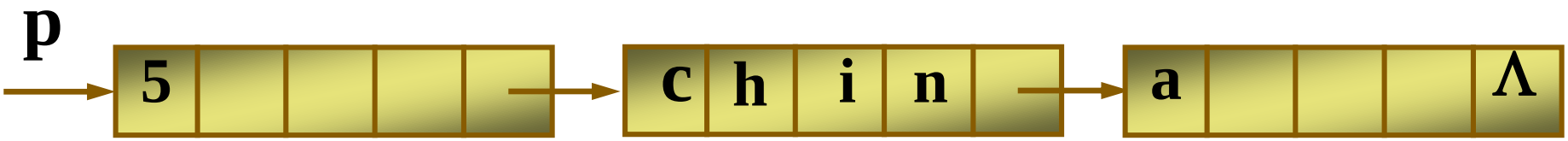
字符串的链接存储

- 串的链接存储：每个结点的结构为： (str, link)，通过链接或指针寻找后继，不要求空间连续。

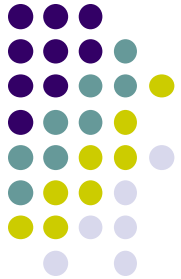
✓ 结点大小为1的链串



✓ 结点大小为4的链串

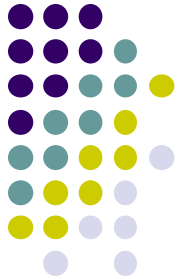


字符串的基本操作



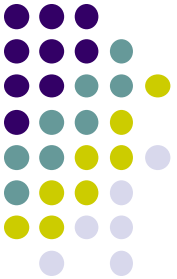
- 求长度
- 比较
- 复制
- 连接(+, strcat)
- 取子串
- 查找
- ……

模式匹配



- **模式匹配问题：在文本文件中查找字符串。**
  - 文本串(text)：文本文件，也称母串
  - 模式串(pattern)：要查找的字符串，也称子串
  - 文本串T，n个字符；模式串P，m个字符
  - 结果：成功，返回匹配成功的起始位置；失败，返回-1
- 应用
  - 文本编辑器（文本文件）中的查找和替换功能
  - DNA中的特定序列搜索
  - ……

朴素模式匹配算法（BruteForce）



T :	a	b	a	a	a	b	a	b
P:	a	b	a	b				
		a	b	a	b			
			a	b	a	b		
				a	b	a	b	
					a	b	a	b

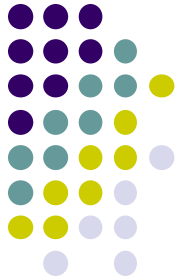
参考代码 I

```
int bfmatch( char s[ ], char p[ ]) {
    int i = 0, j = 0, n=strlen(s), m=strlen(p);
    if ( n < 1 || m < 1 || n < m )  return -1;

    for(i=0; i<=n-m; i++){
        j = 0;
        while(j<m && s[i]==p[j]) i++,j++;
        if(j==m) return i-m;
        i = i- j ;
    }
    return -1;
}
```

参考代码 II

```
int bfmatch( char s[ ], char p[ ]) {
    int i = 0, j = 0, n=strlen(s), m=strlen(p);
    if ( n < 1 || m < 1 || n < m )  return -1;

    while ( i != n && j != m )   {
        if (s[i] == p[j])   ++i , ++j;
        else  i = i - j + 1,  j = 0;
    }
    return j == m ? i - j : -1;
}
```

朴素模式匹配算法分析

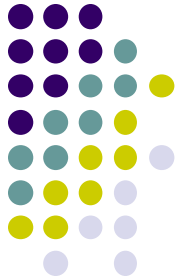


- 在最坏情况下，该算法要匹配n-m+1次，每次匹配做m次比较，因此最坏情况下的比较次数是m(n-m+1)次，时间复杂性为O(mn)。

- 期望时间复杂度也为O(m)。

- 朴素模式匹配算法的优点是简单，缺点是效率低。

一种改进思想



第一次				比较次数				4
S :	a	b	a	a	a	b	a	b
	=	=	=	≠				
P:	a	b	a	b				

第二次				比较次数				1
S：	a	b	a	a	a	b	a	b
		≠						
P:		a	b	a	b			

- $S_1=P_1$， $P_1 \neq P_0$
- 第二次比较必然失败

第三次				比较次数				2
S：	a	b	a	a	a	b	a	b
			=	≠				
P:			a	b	a	b		

- $S_3 \neq P_3$，$P_3 = P_1$
- 第三次比较必然失败

第四次				比较次数				2
T：	a	b	a	a	a	b	a	b
	a	b	a	b				
P:				a	b	a	b	

- 第一次比较完成后，直接进行第四次比较即可，即令朴素算法的指针i=3不变，指针j=0；

KMP算法思想

- 朴素模式匹配算法需要进行5次匹配
- 通过对模式P结构的研究，只需要进行第4、5次比较就可以匹配成功。

- 改进关键：对模式P进行分析可一次向右移多位
- 改进目标：让模式P向右滑得尽可能远

目标形式化

S :	s_0	…	s_{i-j}	s_{i-j+1}		…	s_i	s_{i+1}	…	s_{n-1}
			=	=	=	=	=	≠		
P:			p_0	p_1		…	p_j	p_{j+1}		

P:					p_0	…	p_k	p_{k+1}		

- 寻找最大的k，使得

  $p_0 \cdots p_k = p_{j-k} \cdots p_j$ 且 $p_0 \cdots p_{k+1} \neq p_{j-k} \cdots p_{j+1}$

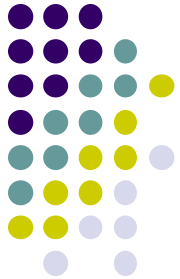
- k是：到 j 字符串的最长公共前缀和后缀 的前缀下标。匹配失败时，j最远能到k。

失败函数

- 定义失败函数**f (j)**，用来确定**k .**

$$f(j)=\begin{cases} k, & k\text{为满足}0\le k<j, \\ & \quad \text{且}p_0p_1\cdots p_k=p_{j-k}p_{j-k+1}\cdots p_j\text{的最大整数} \\ -1, & \text{其他情况} \end{cases}$$

失败函数的计算f(j)



设 $f(0)=-1$ ，设已知 $f(j)=k \Rightarrow f(j+1)=?$

- 如果 $p_{k+1}=p_{j+1}$

p_0	p_1	p_2	p_3	…	p_{j-2}	p_{j-1}	p_j	p_{j+1}	…
a	b	a	a		a	b	a	a	

显然 $f(j+1)=k+1=f(j)+1$

- 如果 $p_{k+1} \neq p_{j+1}$

p_0	p_1	p_2	p_3	…	p_{j-2}	p_{j-1}	p_j	p_{j+1}	…
a	b	a	a		a	b	a	b	

则必有 $f(j+1) \leq f(j)=k$

寻找 h，使它满足 $h<k$，且

$$\left.\begin{array}{l} p_0p_1\cdots p_h = p_{j-h}p_{j-k+1}\cdots p_j \\ p_{j-h}p_{j-h+1}\cdots p_j = p_{k-h}p_{k-h+1}\cdots p_k \end{array}\right\} \Rightarrow p_0p_1\cdots p_h = p_{k-h}p_{k-h+1}\cdots p_k$$

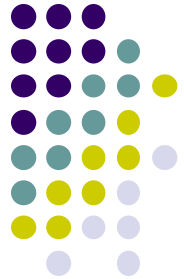
$$\Rightarrow h = f(k)$$

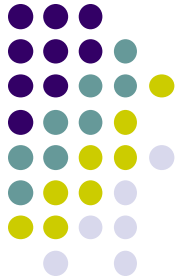
寻找h，满足h<k，且

- 如果 h 不存在，说明 $p_0p_1\cdots p_jp_{j+1}$ 中没有前后相等的子串，因此 $f(j+1)=-1$
- 如果 h 存在，再检验 $p_{h+1}=p_{j+1}$ 是否成立
  - 若成立，说明已找到 $p_0p_1\cdots p_jp_{j+1}$ 中最大的前后相等的子串，即：

$$p_0p_1\cdots p_hp_{h+1} = p_{j-h}p_{j-h+1}\cdots p_{j+1}$$

于是 $f(j+1)=h+1=f(k)+1=f(f(j))+1=f^{(2)}(j)+1$.





- 若 $p_{h+1} \neq p_{j+1}$

  再在 $p_0 p_1 \cdots p_h$ 中寻找最大的前后相等的子串 $p_0 p_1 \cdots p_l$，

  其中 $l = f(h) = f(f(k)) = f^{(2)}(k) = f^{(2)}(f(j)) = f^{(3)}(j)$，

  检验是否有 $p_{l+1} = p_{j+1}$.

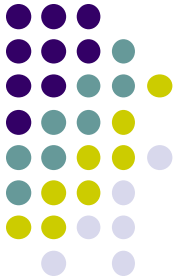
  如此类推，如果找到了某个 t，使得

$$t = f^{(x)}(j) \text{ 且 } p_{t+1} = p_{j+1}$$

则 $f(j+1) = t+1 = f^{(x)}(j)+1.$

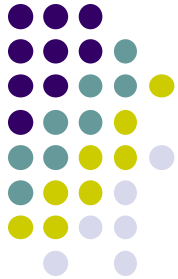
$$f(j+1) = \begin{cases} f^{(x)}(j)+1, & \text{若能找到最小的整数x，使得} p_{f^{(x)}(j)+1} = p_{j+1} \\ -1, & \text{若上述的} x \text{不存在} \end{cases}$$

fail算法参考实现I



```
int f[MAXL];
void cal_fail (char p[])  {
    int i, j, m=strlen(p);
    f[0]=-1;j=-1;
    for(i=1;i<m;i++){
        while(j>=0 && p[j+1]!=p[i]) j=f[j];
        if(p[j+1]==p[i]) j++;
        f[i]=j;
    }
}
```

a	b	c	a	a	a	b	c
p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7



i=0		f(0)= –1		
i=1	j= –1	f(1) = –1		
i=2	j= –1	f(2) = –1		
i=3	j= –1	f(3) = 0		
i=4	j = 0	j = –1	p_4=p_0="a"	f(4) = 0
i=5	j = 0	j= –1	p_5=p_0="a"	f(5) = 0
i=6	j = 0	p_6=p_1="b"	f(6) = 1	
i=7	j = 1	p_7=p_2="c"	f(7) = 2	

KMP算法参考实现I

```
int fast_find(char s[],char p[]) {
    int i, j, n=strlen(s), m=strlen(p);
    if ( n < 1 || m < 1 || n < m )  return -1;
    cal_fail(p);
    j=-1;
    for(i=0;i<n;i++){
        while(j>=0 && p[j+1]!=s[i]) j=f[j];
        if(p[j+1]==s[i]) j++;
        if(j==m-1) return i-j; //p[j+1]==s[i]
    }
    return  -1;
}
```

fail计算实现参考II

```
int f[MAXL];
void cal_fail(char p[]){
    int i=0,j=-1,m=strlen(p); //j是前缀，i是后缀
    f[0] = -1;
    while(i<m-1)
        if(p[i+1]==p[j+1]) f[++i]=++j;
        else if(j > -1) j = f[j]; else f[++i] = -1;
}
```

KMP算法参考实现II

```
int kmp(char s[],char p[]) {
   int i=0,j=0,n=strlen(s),m=strlen(p);

   while(i<n && j<m){
      if(j==-1 || s[i]==p[j]) i++,j++;
      else if(j>0) j = f[j-1] + 1; else j=-1;
   }

   return j==m ? i-j : -1;
}
```

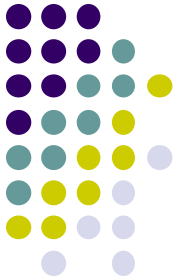
KMP算法分析

- 摊还分析（聚集分析）
  - ✓ i 或者增1，或者 不变
  - ✓ j 或者增1，或者 减少。

- 预处理： 计算fail，O(m)
- KMP算法： O(n)

- 整体时间复杂度O(n+m)

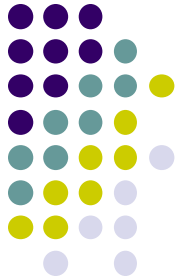
next数组



- **next[j]: j前字符串的最长公共前缀后缀的长度**
- **下标从0开始，j = next[j] 即可**
  - ✓ 有些教材下标从1开始
- **模式串向右移了 j – next[j]位**

参考代码

```
int kmp(char s[],char p[]) {
        int i=0,j=0,n=strlen(s),m=strlen(p);
        if (n < 1 || m < 1 || n < m)  return -1;

        while(i<n && j<m)
           if(j==-1 || s[i]==p[j]) i++,j++;
           else j = next[j] ;

        return j==m ? i-j : -1;
```



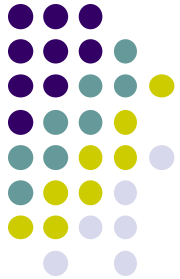
```
void cal_next(char p[]){
        int i=0,j=-1,m=strlen(p); //k是前缀，j是后缀

        next[0] = -1;

        while(i<m-1)
           if(j==-1 || p[i]==p[j]) i++,j++,next[i] = j;
           else j = next [j];
}
```

KMP算法的改进

KMP				比较次数				2
S :	a	b	a	a	a	b	a	b
	a	b	a	b				
P:			a	b	a	b		

实际最远				比较次数				*
S :	a	b	a	a	a	b	a	b
	a	b	a	b				
P:				a	b	a	b	

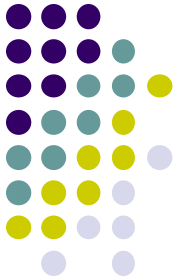
改进措施



- **控制KMP**
  - ✓ j赋值时直接比较1次，p[j] == p[k];
  - ✓ 直观，效率低

- **控制next(推荐)**
  - ✓ if (p[j] != p[k]) next[j] = k; //之前只有这一行
  - ✓ else next[j] = next[k];
  - ✓ 有些书定位为nextval

拓展：模式匹配的其它算法



- **Rabin-Karp**
  - ✓ 编码+hash，可多模匹配
- **Boyer-Moore算法(BM)**
  - ✓ 从后向前
  - ✓ 坏字符规则
  - ✓ 好后缀规则
- **Sunday算法**
  - ✓ 从前向后，失配时考虑串尾。如果该字符没有在模式串中出现，则直接跳过，即移动位数 = 匹配串长度 + 1；否则，其移动位数 = 模式串中最右端的该字符到末尾的距离+1。