数据结构与算法设计

2021-09



课程内容简介

第1章 绪论	第8章 排序与分治	串与串匹配算法
第2章 线性表	第9章 外部排序	红黑树
第3章 栈和队列	第10章 动态规划算法	k-d树
第4章 数组和广义表	第11章 有限自动机	复杂图算法
第5章 树、二叉树、回溯法	第12章 图灵机	文本检索技术
第6章 图与贪心算法	第13章可判定性	分支限界算法
第7章 查找	第14章 时间复杂性	随机化算法
		上下文无关文法



一、什么是串

串是有零个或多个字符组成的有限序列。一般记为:

$$s= a_1 a_2 \cdots a_n$$
,其中 $n>=0$



一、什么是串

$$s = a_1 a_2 \cdots a_n$$

,其中n>=0

s称为串的名, '…'之间的字符序列称为串的值

a可以是字母,数字或其他字符

字符的数目称为串的长度 零个字符的串,称为空串,长度为0



一、什么是串 $s = a_1 a_2 \cdots a_n$,其中n>=0

串中任意连续个字符构成的子序列称为子串 字符在串中的序列中的序号,称为字符的位置 子串第一个字符在主串中的位置,称为子串的位置



a,b都是c和d的子串。

a的长度是3,b的长度是4,c的长度是7,d的长度是8

a在c, d中的位置都是0, b在c中的位置是3, d中的位置是4

当且仅当两个串的值相等,我们说这两个串是相等的

换句话说,当两个串的字符序列和它们在字符序列中的位置都相 等,那么这两个串是相等的。



空格串,由一个或多个空格组成的串,叫做空格串。

s= '

空串是任何串的子串,通常记为∅



一、特殊的线性表

串与线性表极度相似

线性表的操作常常是单个元素为处理对象的;

串的操作常常是以串或子串为处理对象的;

串含有结束标记;



1) 用常量初始化串StrAssgin(&T,chars)

前提: chars是一个字符串常量

功能: 创建一个值等于chars的串T

2) 字符串复制Strcpy(&T,S)

前提:S存在

功能:由串S复制得串T

3) 判別S是否为空串StrEmpty(S)

前提:S必须存在

功能:若S为空串、返回True、否则返回false

4) 字符串比较StrCompare (T,S)

前提:T,S存在

功能: 若S>T,返回值>0,若S与T相等,则返回值==0,

若S<T,返回值<0

5) 求字符串长度StrLength(S)

功能:返回串S的字符个数,即S的长度

6) 清空串ClearStr(&S)

前提:S必须存在

功能:将S清空为空串

7) 字符串连接ConcatStr (&T,S1, S2)

前提: S1、S2存在

功能:连接S1和S2组成新的串T

8) 求子串SubStr(&Sub, S, pos,len)

前提: 串S存在, 0<=pos<=StrLengh(S)且0<=len<=StrLengh(S)-pos

功能:返回串S的第pos个字符起,长度为len的子串Sub



9) 返回子串第一次出现的位置Index(S, T, pos)

前提: S、T必须存在、T是非空串、 0<=pos<=StrLengh(S)

功能:若S中含有和串T相同的子串,则返回它在串S中的第pos个字符之

后第一次出现的位置;否则返回-1

10) 子串替换replace (&S, T, V)

前提:S、T和V都存在、T是非空串

功能:用V替换串S中出现的所有与T相等的不重叠子串



11) 插入子串StrInsert (&S, pos, V)

前提: S和T存在, 0<=pos<=StrLengh(S)

功能:在串S的第pos个位置之前插入串T

12) 删除子串StrDelete (&S, pos, len)

前提: S存在, O<=pos<=StrLengh(S)-len

功能: 从S中删除第pos个字符起长度为len的子串

13)释放串的存储空间(&S)

前提:S存在

功能:串S被销毁



- 二、串的物理实现方式
- ·順序结构(数组,C语言)
- 链表方式(略)



4.3.1 串的模式匹配算法: 查找子串定位的Index(S,P,pos)_暴力法

```
int i=pos , j=0;
while(i<strlen(S)&&j<strlen(P))
{
    if ( S[i] == P[j] ){    i++; j++; }
    else { i=i-j+1; j=0; }
}
if (i>=strlen(S)) printf( "%d" ,-1);//匹配不成功
else printf("%d",i-j);//匹配成功
```

最好的匹配成功的情况是m, 最好的匹配不成功的情况是n,

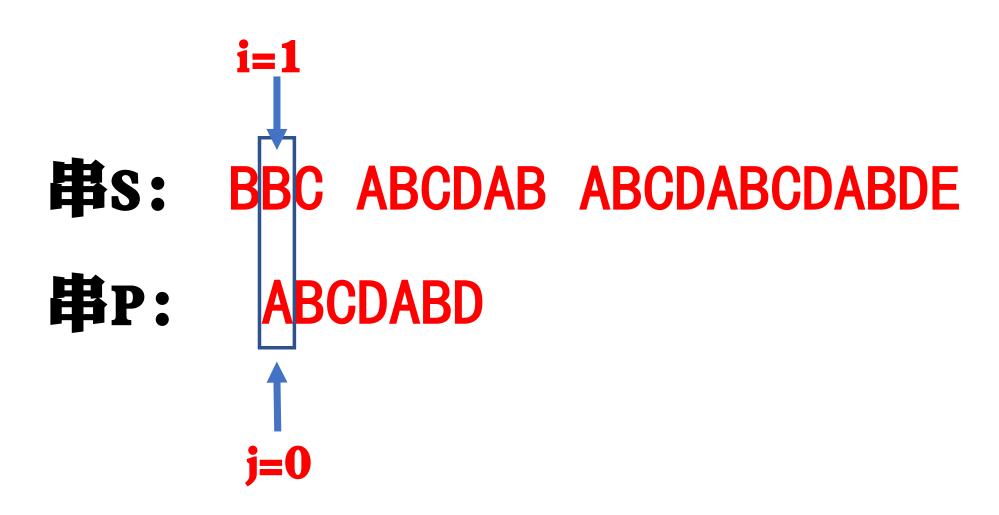


4t 最坏的情况是n*m

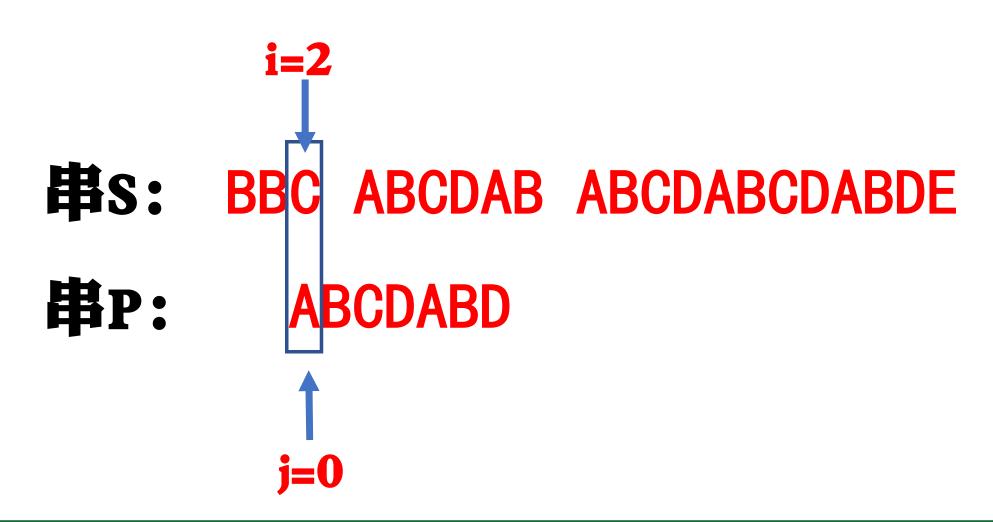




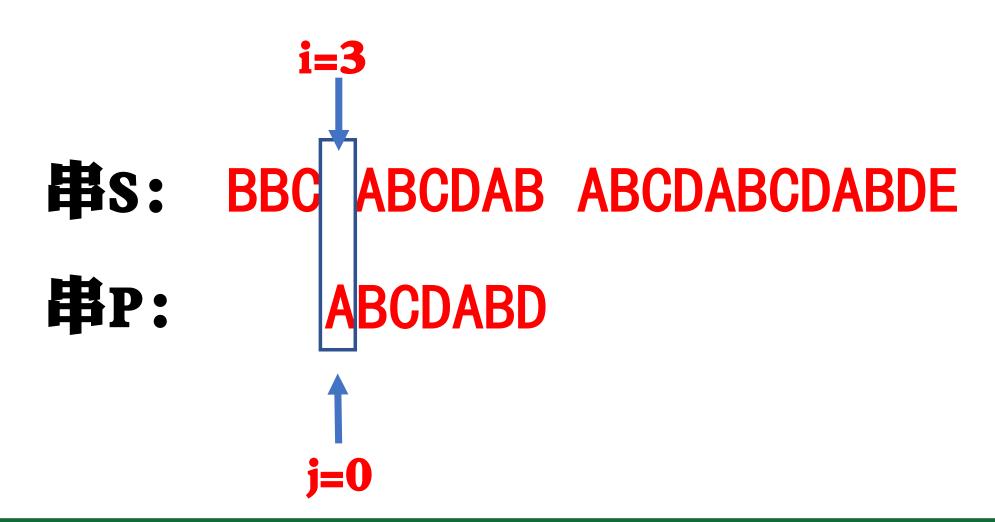




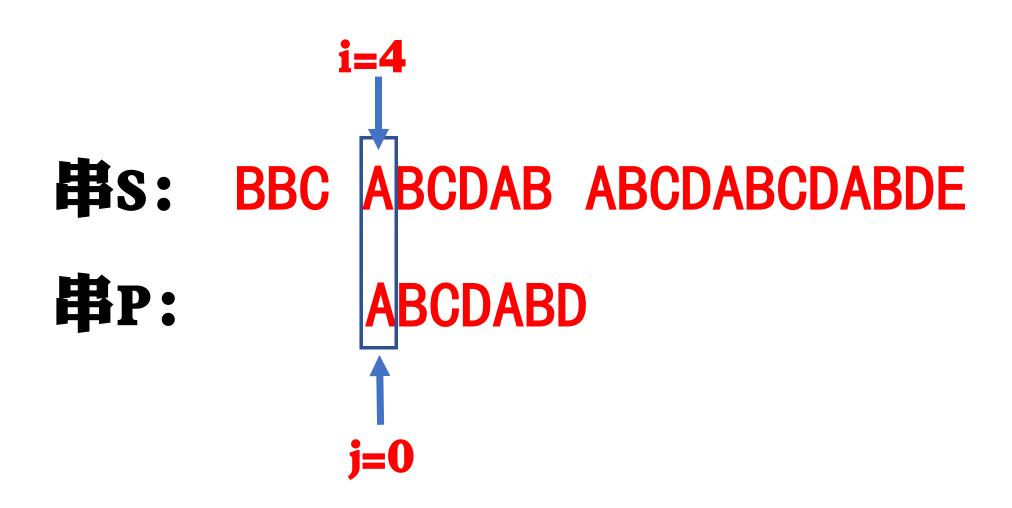




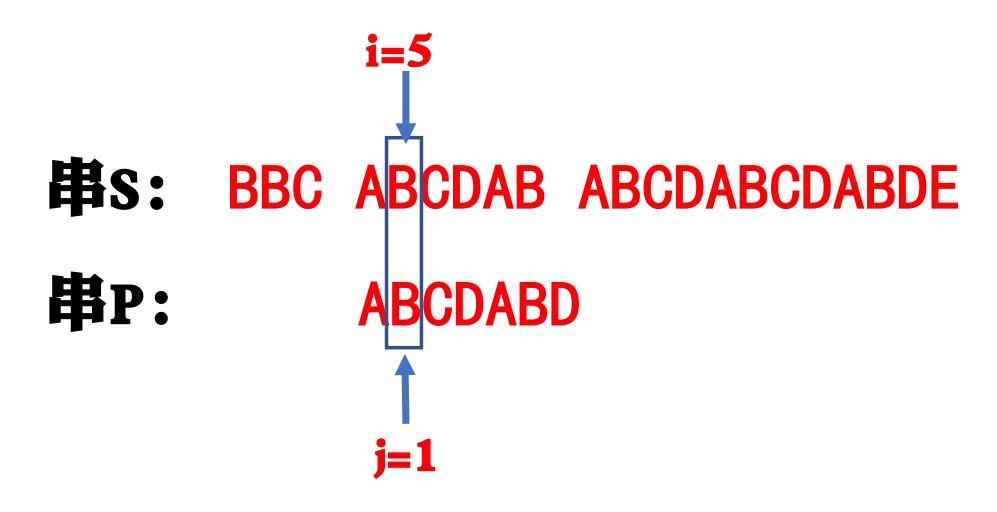




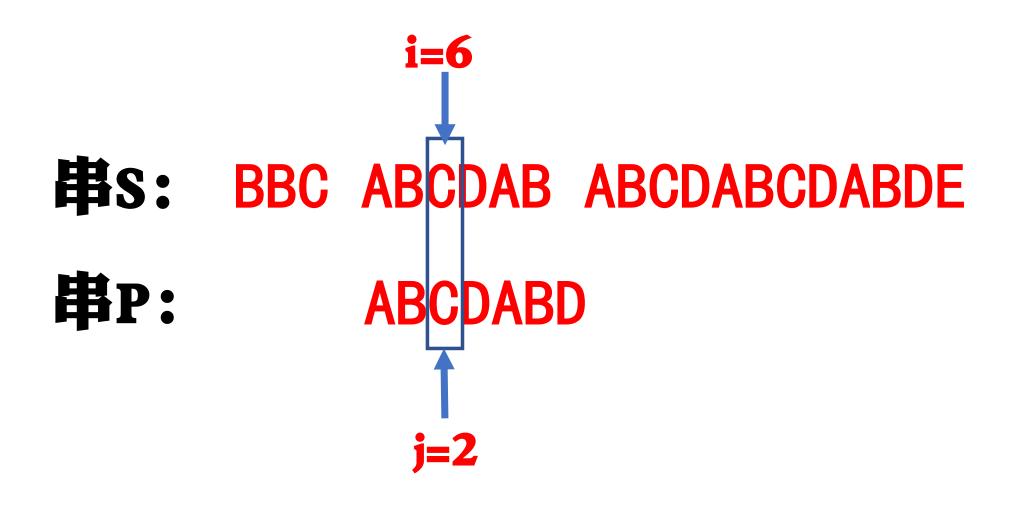




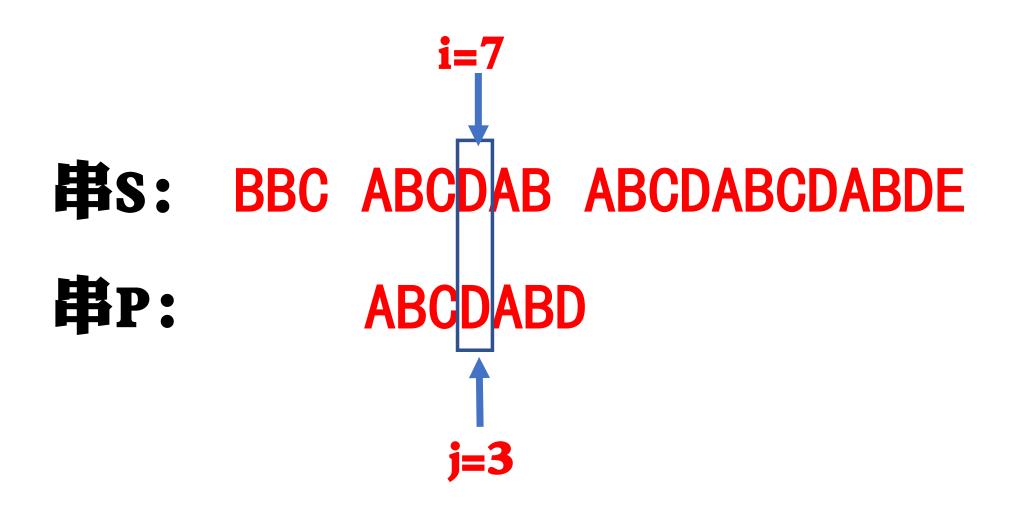




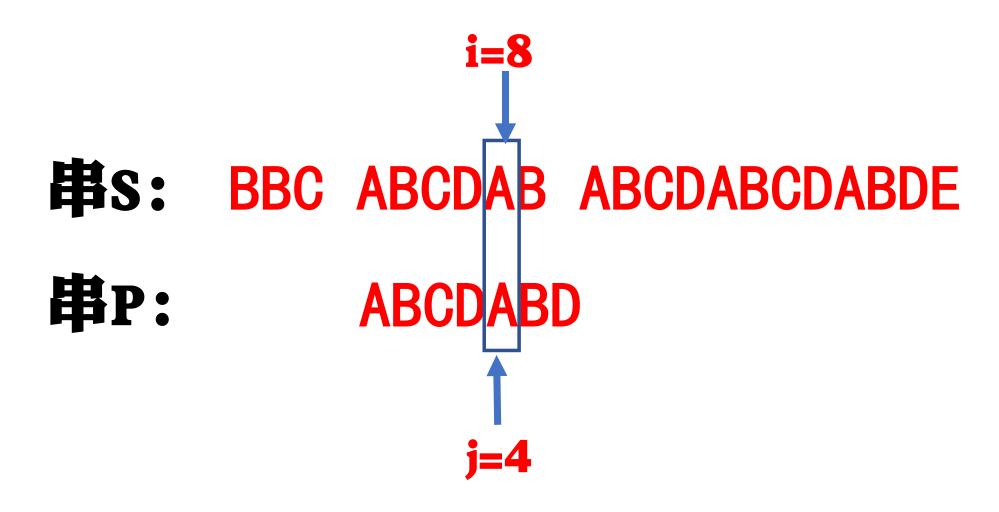




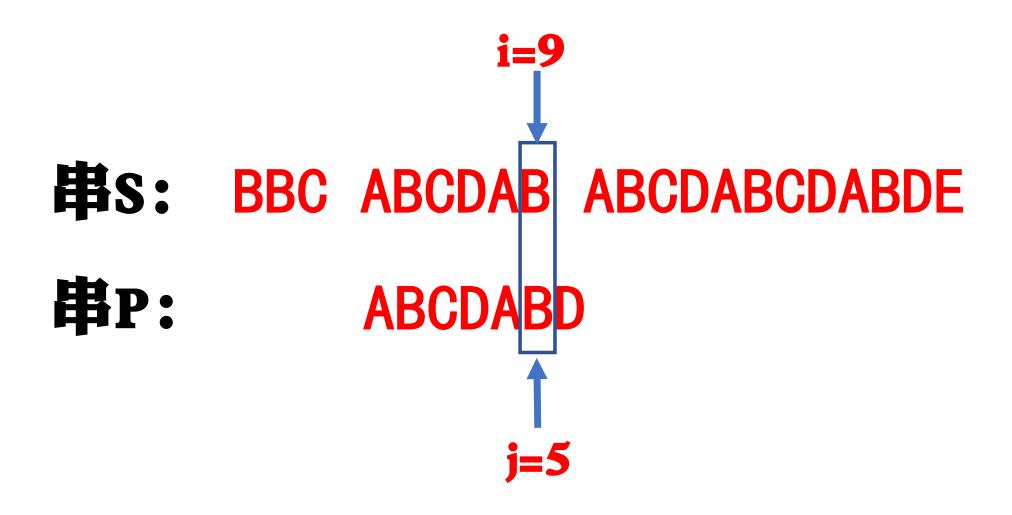




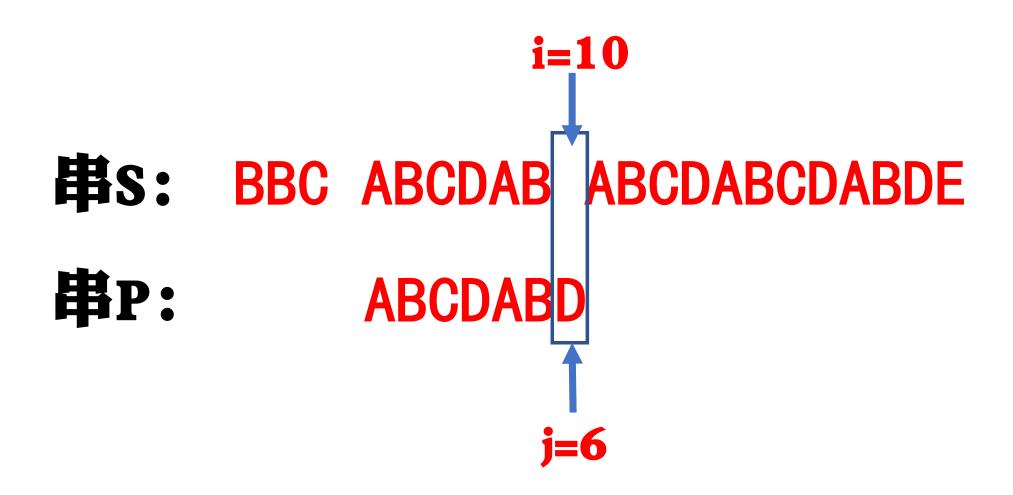




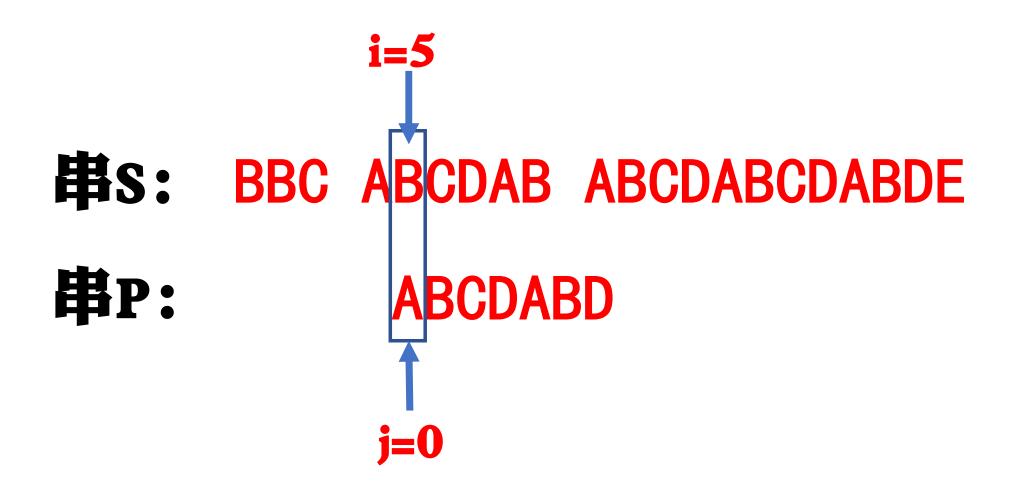














一、串的模式匹配算法

4.3.1 串的模式匹配算法: KMP算法

算法由D.E.Knuth、J.H.Morris和V.R.Pratt,共同发现

利用之前已经部分匹配这个有效信息,保持:不回溯,通过修改的位置, 让模式串尽量地移动到有效的位置。



KMP的基本思想

假设现在文本串S匹配到 i 位置,模式串P匹配到 j 位置:

如果当前匹配失败:

在i保持不变的情况下,i应该从哪里开始进行匹配?



KMP的基本思想

假设现在文本串S匹配到 i 位置,模式串P匹配到 i 位置:

如果当前匹配失败:

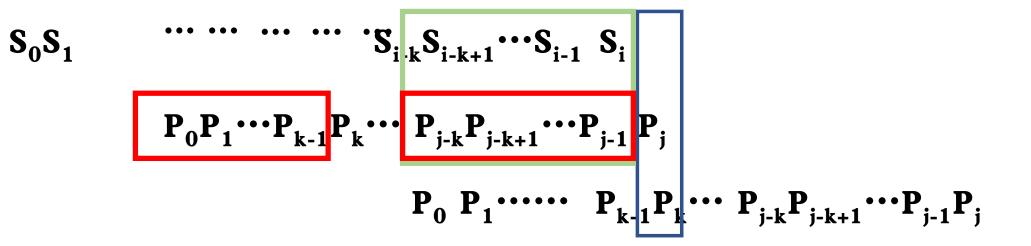
在i保持不变的情况下,i应该从哪里开始进行匹配?

假定i位置的字符,应该与P串中的第k个字符做匹配,那么:



 S_0S_1

$$\mathbf{P}_0\mathbf{P}_1\cdots\mathbf{P}_{k-1}\mathbf{P}_k\cdots\mathbf{P}_{j-k}\mathbf{P}_{j-k+1}\cdots\mathbf{P}_{i-1}$$





定义Next数组,用于存储当i与j匹配失败的时候,下一步应该让i去匹配P[k] 字符、关键是求得这个k:

问题变成了、求解k的过程:

(1) 给出k的定义求解:



模式串的各个子串	前缀	后缀	最大公共元素长度	
Α	空	至	0	
AB	А	В	0	
ABC	A,AB	C,BC	0	
ABCD	A,AB,ABC	D,CD,BCD	0	
ABCDA	A,AB,ABC,ABCD	A,DA,CDA,BCDA	1	
ABCDAB	A,AB,ABC,ABCD,ABCDA	B,AB,DAB,CDAB,BCDAB	2	
ABCDABD	A,AB,ABC,ABCD,ABCDA ABCDAB	D,BD,ABD,DABD,CDABD BCDABD	0	



next 数组考虑的是除当前字符外的最长相同前缀后缀,所以通过第 ①步骤求得各个前缀后缀的公共元素的最大长度后,只要稍作变形 即可:将第①步骤中求得的值整体右移一位,然后初值赋为-1,如 下表格所示:

模式串	A	В	C	D	A	В	D
最大公共元素 长度	0	0	0	0	1	2	0
j	0	1	2	3	4	5	6
next 数组的值: next[j]	-1	0	0	0	0	1	2



(2) next数组的使用(初始i=0, j=0):

在匹配过程中:

- (0) 比较S_i和P_i
- (1) 若S_i==P_i,则i++,j++,继续(0)匹配S_i和P_i;
- (2) 若S_i!=P_i,则退回到k(next[j])的位置,即j赋值为k(j<-k),再比较:

若S;与P,相等:则i++,j++;

若S;与P,不相等,则继续退回到next[j]的位置,而此时j==k,

next[j]变相等于next[next[j]]、依次类推下去、直到:

a:退回到next[.....next[j]];

b:退回到0,则开始比较S_{i+1}与P₀



(3) 给出k的递归求解过程:

 \Rightarrow : next[0]=-1,next[1]=0;

设: next[j]=k,即表示当;字符"失配"时,应退到第k个元素,因为: 有" $P_0P_1....P_{k-1}$ " == " $P_{i-k}P_{i-k+1}......P_{i-1}$ "

接下来、考察next[i+1]:

 ${\bf Z}_k=P_i$,则有" $P_0P_1....P_{k-1}P_k$ " == " $P_{i-k}P_{i-k+1}......P_{i-1}P_i$ ",显然也 有 next[j+1]=k+1,

若P,!=P,,则有两种情况:

- a) next[j+1]=next[k]+1, (存在一个 $P_i==P_k$,)
- b) next[j+1]=0、要么从0开始比较

next 数组递推过程(初始值)

模式串	A	В	C	D	A	В	D
j	0	1	2	3	4	5	6
next 数组的值: next[j]	-1	0					

根据P₀, P₁, 求next[2]的值: P_i是'B', k是0, P_k是'A', 所以next[2]为0



next 数组递推过程 (next[2]=0)

模式串	A	В	C	D	A	В	D
j	0	1	2	3	4	5	6
next 数组的值: next[j]	-1	0	0				

根据P₀, P₁, P₂求next[3]的值: P_i是 'C', k是0, P_k是 'A', 所以next[3]为0



next 数组递推过程 (next[3]=0)

模式串	A	В	C	D	A	В	D
j	0	1	2	3	4	5	6
next 数组的值: next[j]	-1	0	0	0			

根据P₀, P₁, P₂, P₃求next[4]的值: P_i是 'D', k是0, P_k是 'A', 所以next[4]为0



next 数组递推过程 (next[4]=0)

模式串	A	В	C	D	A	В	D
j	0	1	2	3	4	5	6
next 数组的值: next[j]	-1	0	0	0	0		

根据P₀, P₁, P₂, P₃, P₄, 求next[5]的值: P_j是 'A', k是0, P_k是 'A', 所以next[5]为next[4]+1



next 数组递推过程 (next[5]=1)

模式串	A	В	C	D	A	В	D
j	0	1	2	3	4	5	6
next 数组的值: next[j]	-1	0	0	0	0	1	

根据 P_0 , P_1 , P_2 , P_3 , P_4 , P_5 求next[6]的值: P_i是 'B', k是1, P_k是 'B', 所以next[6]为next[5]+1



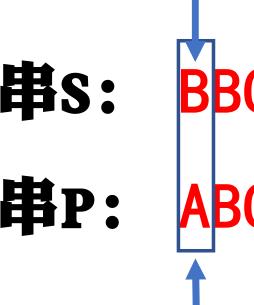
next 数组递推过程 (next[6]=2)

模式串	A	В	C	D	A	В	D
j	0	1	2	3	4	5	6
next 数组的值: next[j]	-1	0	0	0	0	1	2



```
int j,k;
k=-1; j=0;
next[0]=-1;
while(j<S.length){
      if (k==-1 | || (S[j]==S[k]) {|}
             j++; k++; next[j]=k; }
       else
             k=next[k];
```

模式串	Α	В	С	D	A	В	D
最大长度值	0	0	0	0	1	2	0
next 数组	-1	0	0	0	0	1	2



i=0

j=0

BBC ABCDAB ABCDABCDABDE

ABCDABD



		模式串	A	В	C	D	A	В
		最大长度值	0	0	0	0	1	2
	i=1	next 数组	-1	0	0	0	0	1
	<u> </u>							
串S:	BBC A	BCD/	B	ABC	DAE	3CD	ABC)E
串P:	ABCD	ABD						
	†							



j=0

0

2

		模式串	А	В	С	D	A	В
		最大长度值	0	0	0	0	1	2
	i=2	next 数组	-1	0	0	0	0	1
串S:	BBC	ABCDA	λB	ABC	DAE	3CD	ABD	E
串P:	ABO	CDABD						
	A							

j=0

0

2

模式串	A	В	C	D	A	В	D
最大长度值	0	0	0	0	1	2	0
next 数组	-1	0	0	0	0	1	2

串S: BBC

ABCDAB ABCDABCDABDE

串P:

ABCDABD



i=3

		模式串	А	В	С	D	А	В
		最大长度值	0	0	0	0	1	2
	i=4	next 数组	-1	0	0	0	0	1
串S:	BBC	ABCD/	AB	ABC	DAE	3CD	ABC)E
串P:		ABCD <i>A</i>	ABD					
	j:	=0						



0

2

模式串	А	В	С	D	А	В	D
最大长度值	0	0	0	0	1	2	0
next 数组	-1	0	0	0	0	1	2

#S: BBC ABCDAB ABCDABCDABDE

串P:

ABCDAB ABCDABCDA

i=10

模式串	A	В	С	D	А	В	D
最大长度值	0	0	0	0	1	2	0
next 数组	-1	0	0	0	0	1	2

串S:

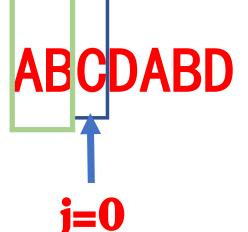
串P:

i=10BBC ABCDAB ABCDABCDABDE **ABCDAED**

模式串	A	В	С	D	А	В	D
最大长度值	0	0	0	0	1	2	0
next 数组	-1	0	0	0	0	1	2



串P:



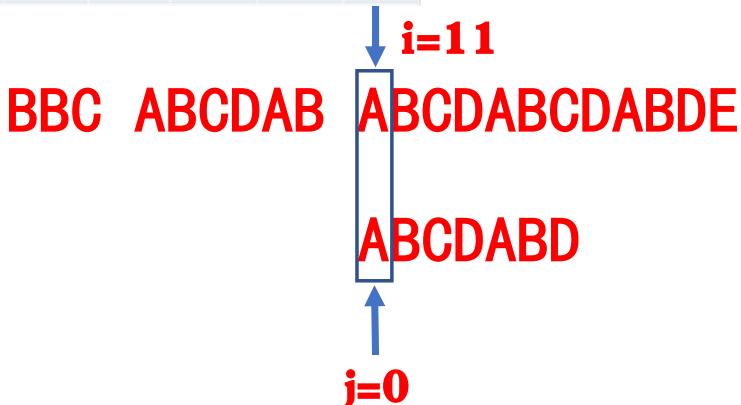
i=10



模式串	A	В	С	D	A	В	D
最大长度值	0	0	0	0	1	2	0
next 数组	-1	0	0	0	0	1	2

串S:

串P:

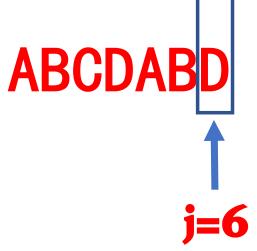




模式串	A	В	С	D	A	В	D
最大长度值	0	0	0	0	1	2	0
next 数组	-1	0	0	0	0	1	2

#S: BBC ABCDAB ABCDABCDABDE

串P:

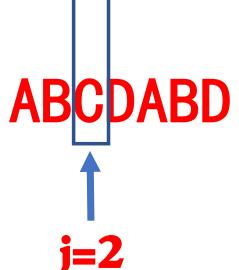




模式串	A	В	С	D	A	В	D
最大长度值	0	0	0	0	1	2	0
next 数组	-1	0	0	0	0	1	2

#S: BBC ABCDAB ABCDABCDABDE

串P:





模式串	A	В	С	D	A	В	D
最大长度值	0	0	0	0	1	2	0
next 数组	-1	0	0	0	0	1	2



串S: BBC ABCDAB ABCDABCDABD

串P:

ABCDABD
ABCDABD
j=6

i始终没有回溯,一直向右,提高了效率



北京理工大学

梅以明理 学以新己

KMP改进方法



◎ 北京理工大学

串S: abacababc



↓↓↓↓↓ 串T: abab

	a	ь	a	ь
索引值	0	1	2	3
最大前缀后缀	0	0	1	2
next[j]	-1	0	0	1
是否满足	初始值 无须优化	P[1]!=P[next[1]]	P[2] == P[next[2]]	p[3]==p[next[3]]
是否优化	初始值 无须优化	不优化	需要优化	需要优化
优化的next[j]	-1	0	next[2]=next[next[2]]=next[0]=-1	next[3]=next[next[3]] =next[1]=0

串S: abacababc

串T: abab

	a	ь	a	ь
索引值	0	1	2	3
最大前缀后缀	0	0	1	2
next[j]	-1	0	0	1
优化的next[j]	-1	0	-1	0



重复比较

串S: abacababc

abacababc

abab

串T: abab

按照next值,应该移到1号位置

	a	b	a	b
索引值	0	1	2	3
最大前缀后缀	0	0	1	2
next[j]	-1	0	0	1
优化的next[j]	-1	0	-1	0



串S: abacababc

串T: abab

串S: abacababc

abab

按照优化的next值,应该移到0号位置

	a	b	a	b
索引值	0	1	2	3
最大前缀后缀	0	0	1	2
next[j]	-1	0	0	1
优化的next[j]	-1	0	-1	0



```
int j,k;
 j=0;k=-1;
 nextval[0]=-1;
 while(j<S.length-1){
        if (k==-1 || S[j]==S[k]){
                j++;k++;
                 if (S[j])!=S[k]{ nextval[j]=k; }
                else nextval[j]=nextval[k];
         else
            k=nextval[k];
```

1/2 11 114 10 3 1 1 1 1 1

・KMP的匹配是从模式串的开头开始匹配的,而1977年,德克萨斯 大学的Robert S. Boyer教授和J Strother Moore教授发明了一种新 的字符串匹配算法: Boyer-Moore算法, 简称BM算法。该算法从 模式串的尾部开始匹配,且拥有在最坏情况下O(N)的时间复杂度。 在实践中、比KMP算法的实际效能高。

- ·BM算法定义了两个规则:
- ・坏字符规则:当文本串中的某个字符跟模式串的某个字符不匹配 时,我们称文本串中的这个失配字符为坏字符,此时模式串需要 向右移动、移动的位数 = 坏字符在模式串中的位置 - 坏字符在模 式串中最右出现的位置。此外,如果"坏字符"不包含在模式串之 中、则最右出现位置为-1。
- •好后缀规则:当字符失配时,后移位数 = 好后缀在模式串中的位 置 - 好后缀在模式串上一次出现的位置,且如果好后缀在模式串 中没有再次出现、则为-1。



#S: HERE IS A SIMPLE EXAMPLE

串P: EXAMPLE

"文本串"与"模式串"头部对齐,从尾部开始比较。"S"与"E"不匹配。这时, "S"就被称为"坏字符"(bad character),即不匹配的字符,它对应着模式 串的第6位。且"S"不包含在模式串"EXAMPLE"之中(相当于最右出现位置 是-1),这意味着可以把模式串后移6-(-1)=7位,从而直接移到"S"的后一 位。



串S: HERE IS A SIMPLE EXAMPLE

串P: EXAMPLE

从尾部开始比较。"P"与"E"不匹配,坏字符,它对应着模式串的第6位。 但"P"包含在模式串"EXAMPLE"之中(出现位置是4),这意味着可以把模式串后移6-4=2位,从而直接后移2位,两个"P"对齐。



#S: HERE IS A SIMPLE EXAMPLE

串P:

"P"与"P"匹配。这时,"P"就被称为"好字符",即所有尾部匹配的字符串。注意,"MPLE"、"PLE"、"LE"、"E"都是好后缀。

EXAMPLE



串S: HERE IS A SIMPLE EXAMPLE EXAMPLE EXAMPLE

串P:

"I"与"A"不匹配。这时,按照坏字符规则,向右移3位(2-(-1))。 所有的"好字符",即所有尾部匹配的字符串。注意,"MPLE"、 "PLE"、"LE"、"E"都是好后缀。后移位数 = 好后缀在模式串中的位置 - 好后缀在模式串中上一次出现的位置,且如果好后缀在模式串中没有再次出现,则为-1。所有的"好后缀"(MPLE、PLE、LE、E)之中,只有"E"在"EXAMPLE"的尾部和头部出现,所以后移6-0=6位。

串: HERE IS A SIMPLE EXAMPLE EXAMPLE EXAMPLE

串P:

可以看出,"坏字符规则"只能移3位,"好后缀规则"可以移6位。每次后移这两个规则之中的较大值。这两个规则的移动位数,只与模式串有关,与原文本串无关。

#S: HERE IS A SIMPLE EXAMPLE

串P:

EXAMPLE

"P"与"E"不匹配。这时,"P"就被称为"坏字符"(bad character),它对应着模式串的第6位。且"P"包含在模式串"EXAMPLE"之中(最右出现位置是4),这意味着可以把模式串后移6-(4)=2位。



串S: HERE IS A SIMPLE EXAMPLE EXAMPLE EXAMPLE

"E"与"E"匹配。这时,"E"就被称为"好字符"(good character),它对应着模式串的第6位。"EXAMPLE""XAMPLE" "AMPLE""MPLE"、"PLE"、"LE"、"E"都是好后缀。



Sunday算法

BM算法虽然通常比KMP算法快,但BM算法也还不是现有字 符串查找算法中最快的算法,最后再介绍一种比BM算法更快的查 找算法即Sunday算法。

Sunday算法由Daniel M.Sunday在1990年提出,它的思想跟 BM算法很相似:

只不过Sunday算法是从前往后匹配,在匹配失败时关注的是 文本串中参加匹配的最末位字符的下一位字符。

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



串S: substring searching algorithm

串P: search

结果发现在第2个字符处发现不匹配,不匹配时关注文本串中参加匹配的最末位字符的下一位字符,即标粗的字符 i,因为模式串search中并不存在i,所以模式串直接跳过一大片,向右移动位数 = 匹配串长度 + 1 = 6 + 1 = 7,从 i 之后的那个字符(即字符n)开始下一步的匹配。



串S: substring searching algorithm search

第一个字符就不匹配,再看文本串中参加匹配的最末位字符的下一位字符,是'r',它出现在模式串中的倒数第3位,于是把模式串向右移动3位(r 到模式串末尾的距离 + 1 = 2 + 1 = 3),使两个'r'对齐



串S: substring searching algorithm search

匹配成功



本章学习要点

- 1. 掌握串类型的特点,并能在相应的应用问题中正确选用 它们。
- 2. 熟练KMP实现方法
- 3. 了解BM算法和Sunday算法

