数据结构与算法

串与KMP

计算机学院

朱晨阳 副教授

zhuchenyang07@nudt.edu.cn

串的基本概念

- 串是字符的有限序列，也称字符串。串是一种线性表，每个结点的数据为一个字符。
- 串广泛应用于输入输出、文本编辑、信息搜索等。搜索引擎的核心技术是高效快速的串匹配算法。

串的基本概念

- C语言中的字符串如：

(1) "this is a sample string",

(2) "a sample string",

(3) "The emergency phone number is 119",

(4) " ",

(5) ""。

- 串中任意个连续的字符组成的子序列称为该串的子串。空串是任何串的子串。任意串都是其自身的子串。串S的子串中除了其自身外，都是S的真子串。

串的存储

- 顺序存储是串的最常用的存储方式。
- C/C++中，每个字符占用一个字节，最后附加**0x00**（即字符'**\0**'）表示字符串的结束。顺序存储的串在删除字符和插入字符的操作时，都要移动字符。

串的实现

```
#define MAX_STRING_SIZE 1024
struct CMyString
{
    int length;
    char str[MAX_STRING_SIZE+1];
}
void InitCMyString(CMyString* CS, char* s)
{   char* p1, *p2;
    for(CS->length=0, p1=CS->str, p2=s;  *p2;  CS->length++)
        *p1++=*p2++;
    *p1=0;
}
```

串的实现

- memcpy(str1，str2，length)

```
// 将字符串复制到数组 dest 中
#include <stdio.h>
#include <string.h>
 int main ()
{
        const char src[50]="http://www.gotonudt.cn";
        char dest[50];
        memcpy(dest, src, strlen(src)+1);
        printf("dest = %s\n", dest);
        return(0);
}
```

串的实现

```
#include <stdio.h>
#include<string.h>
int main()
{
        char *s="http://www.gotonudt.cn";
        char d[20];
        memcpy(d, s+11, 8);
    // 从第 11 个字符(g)开始复制，连续复制 8 个字符(gotonudt)
        d[8]='\0’;
        printf("%s", d); return 0;
}
```

串的实现

```
char* GetString(CMyString* CS)
{   char* tmpStr=(char*)malloc(sizeof(char)*(CS->length+1));
    memcpy(tempStr, CS->str, CS->length+1);
    return tmpStr;
}

//{p[0],p[1],…,p[n-1]},   {s[0],s[1],…,s[m-1]} →
//{p[0], p[1], …, p[n-1], s[0], s[1], …, s[m-1]}
void Concatenate(CMyString *CS, CMyString * s)
{   if(CS->length+s->length<=MAX_STRING_SIZE)
    {  memcpy(CS->str+CS->length, s->str, s->length+1);
       CS->length+=s->length;
    }
    else printf("error: string length overflow!\n")
}
```

串的实现

```
//{p[0], …, p[n-1]} →
//{p[0],…,p[pos-1],    p[pos+len], …, p[n-1]}
void DeleteCS(CMyString* CS, int Pos, int len)
{    int rlen=len; //需要删除的子串的长度
     if(pos+rlen>CS->length)
              rlen=CS->length-pos;
     CS->length = CS->length - rlen;
     memcpy(CS->str+pos, CS->str+pos+rlen,
                CS->length-pos+1);
}
```

串的实现

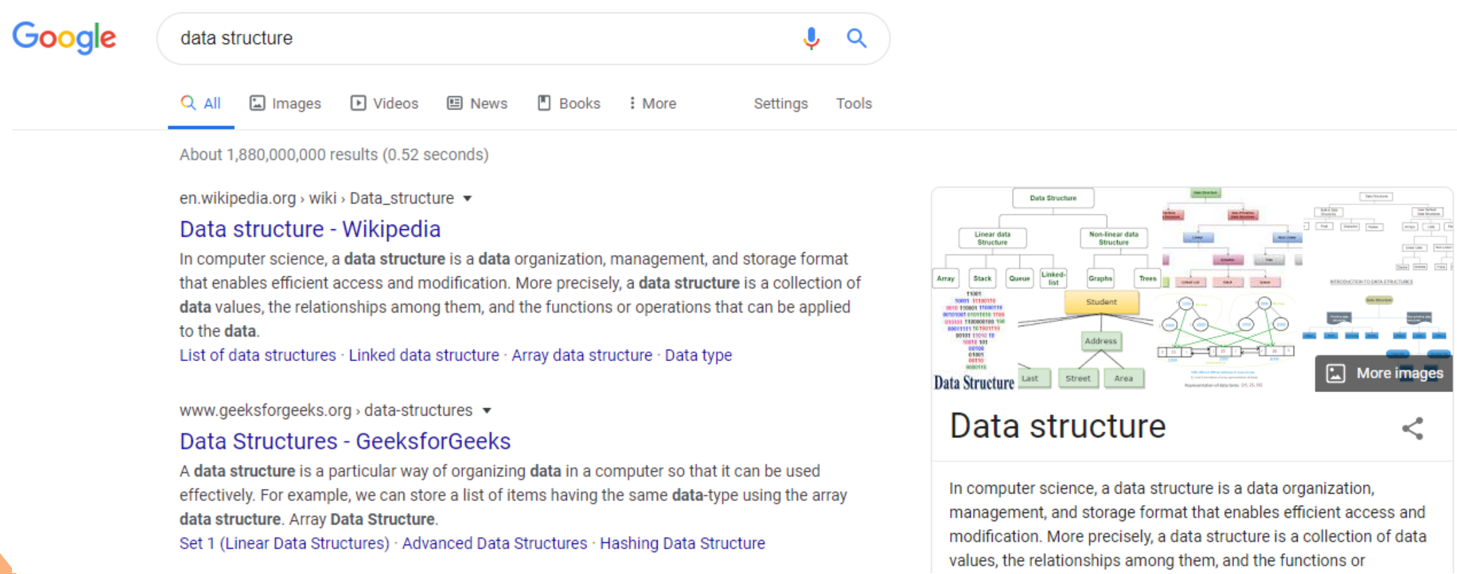
```
//{p[0], …, p[n-1]},  {s[0], …, s[m-1]} →
//{p[0], …, p[pos-1], s[0], …, s[m-1], p[pos], …, p[n-1]}
void Insert(CMyString* CS, int pos, CMyString *s)
{   if(CS->length+s->length<=MAX_STRING_SIZE)
    {
        memcpy(CS->str+pos+s->length,CS->str+pos,
                  CS->length-pos+1);
        memcpy(CS->str+pos, s->str, s->length);
        CS->length+=s->length;
    }
    else printf("error: string length overflow!\n");
}
```

串的实现

```
//get {p[pos], …, p[pos+len-1]}
CMyString SubString(CMyString *CS, const int pos, const int len)
{   int rlen=len;
    CMyString* tmpStr=(CMyString*)malloc
                                   (sizeof(CMyString));
    InitCMyString(tmpStr, “”);
    if(pos+len>CS->length)   rlen=CS->length-pos;
    memcpy(temStr->str, CS->str+pos, rlen);
    tmpStr->length=rlen;
    tmpStr->str[tmpStr->length]=0;
    return *tmpStr;
}
```

串的匹配

- 串匹配问题：给定字符串P和T，从T中找出与P相同的第一个子串或所有子串
- 搜索引擎



定义p[a : b]

- 定义p[-1]为空字符.
- 定义p[a:b]，
  - a<=b时，p[a:b]表示序p[a]p[a+1]p[a+2]...p[b]
  - a>b时，p[a:b]表示空序列
  - 例如p="string", 则p[2:4]="rin", p[2:1]="".
- p[a:b]=q[c:d]当且仅当
  - a>b且c>d，即都为空序列，或者
  - a<=b 且 c<=d 且 b-a=d-c 且

p[a]=q[c] , p[a+1]=q[c+1], ..., p[b]=q[d]

朴素的串匹配算法

- 朴素的串匹配算法:
- 将P中的字符依次与T中的字符进行比较。
- 设T=t[0:n-1], P=p[0:m-1], m<=n.
- 从T的最左端开始进行比较
  - 第**1**趟:如果**t[0:m-1]=p[0:m-1]**, 则匹配成功
  - 第**2**趟:如果**t[**
  - 第**i**趟:如果**t[i** 功

顶多多少趟? 最多进行多少次匹配？
时间复杂度为多少？

朴素的串匹配算法

T="aaabbaaaba", P="aaaba"

a	a	a	b	b	a	a	a	b	a
a	a	a	b	a					
	a	a	a	b	a				
		a	a	a	b	a			
			a	a	a	b	a		
				a	a	a	b	a	
					a	a	a	b	a

- 朴素的串匹配算法每趟最多比较 m次，最多 n-m+1 趟，总的比较次数最多为 m(n-m+1)，所以时间复杂度为 O(mn) 。
- 朴素的匹配算法效率低，实际的应用中很少采用。

朴素的串匹配算法

- 哪些可以跳过?
- 第1趟比较发现:
  - P[0:3]=T[0:3],P[4]≠T[4]
- 基于第1趟的比较结果，结合P的特点，是否可以断定第2趟一定会失败?为什么?

T="aaabbaaaba", P="aaaba"

	a	a	a	b	b	a	a	a	b	a
1	a	a	a	b	a					
2		a	a	a	b	a				
3			a	a	a	b	a			
4				a	a	a	b	a		
5					a	a	a	b	a	
6						a	a	a	b	a

KMP匹配算法

- KMP匹配算法是由Knuth,Morris和Pratt提出的一种快速的串匹配算法。KMP算法考虑:

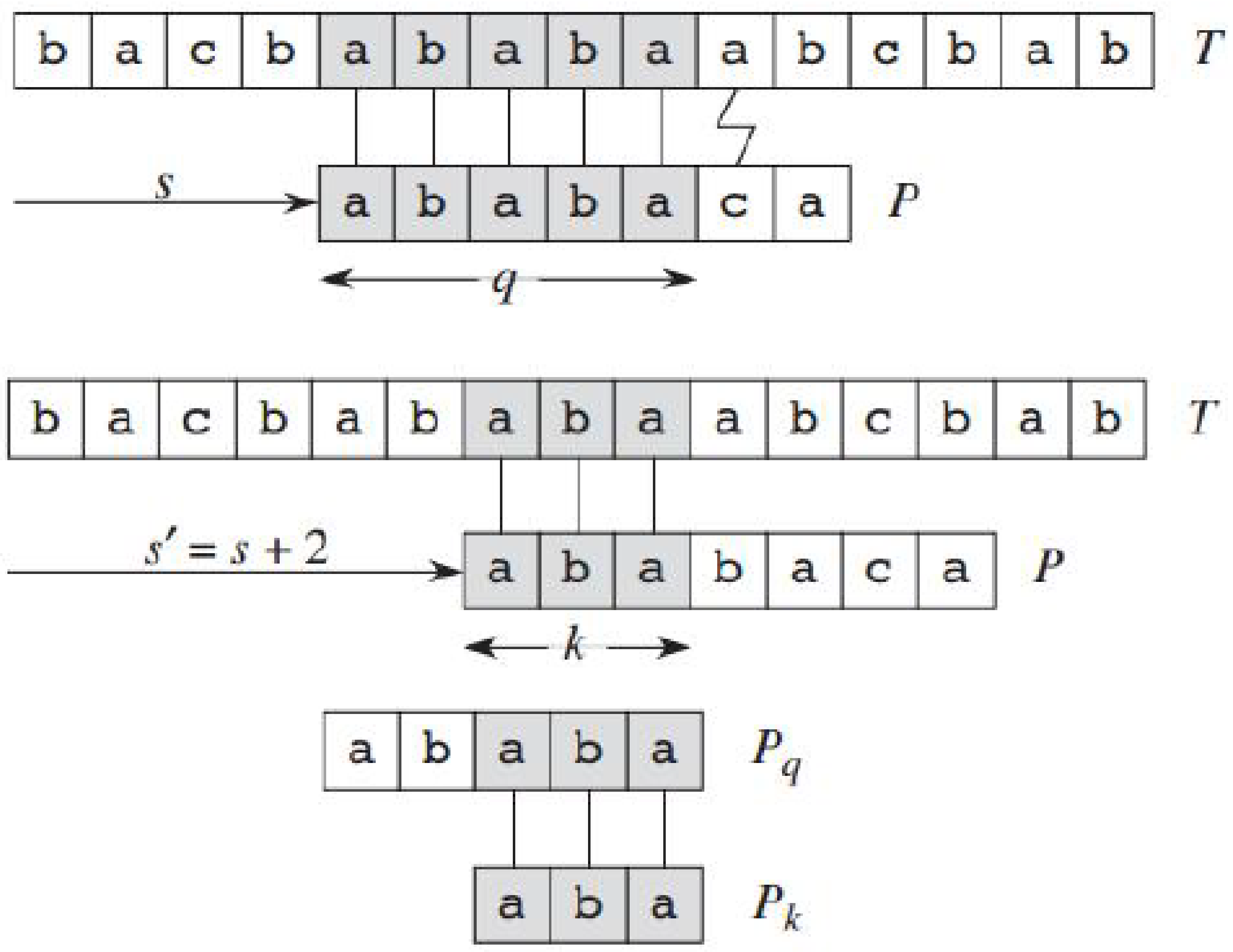
(1)当匹配失败时，应该将P右移多少个字符；

(2) P右移后，应该从P中的哪个字符开始比较。

	t[j-i]	t[j-i+1]	t[j-i+2]	…	t[j-k]	…	t[j-1]	t[j]	
	p[0]	p[1]	p[2]	…	p[i-k]	…	p[i-1]	p[i]	
	YES	YES	YES	YES	YES	YES	YES	NO	
					p[0：x-1]			p[x]	

P[i]处发生匹配失败时，P右移i-x个字符，如何确定x？

KMP匹配算法


b a c b a b a b a a b c b a b
T
s
a b a b a c a
P
q
b a c b a b a b a a b c b a b
T
s′ = s + 2
a b a b a c a
P
k
a b a b a
P_q
a b a
P_k

KMP匹配算法

t[j-i]	t[j-i+1]	t[j-i+2]	…	t[j-k]	…	t[j-1]	t[j]
p[0]	p[1]	p[2]	…	p[i-k]	…	p[i-1]	p[i]
	p[0]	p[1]	…		…	p[i-2]	p[i-1]
		p[0]	…		…	p[i-3]	p[i-2]
			…	p[0]	…	p[k-1]	p[k]

KMP匹配算法

- 在p[i]与t[j]匹配时发生失败，则：
  - p[0:i-1]=t[j-i : j-1]， p[i] ≠ t[j]
- 右移p，使p[k]移到t[j]处，即p右移i-k位
  - 若p[0:k-1]≠p[i-k:i-1] 或 p[k]=p[i](即p[k]=p[i]≠ t[j])，则一定失败
  - 若p[0:k-1]=p[i-k:i-1] 且 p[k]≠p[i]，则还有可能继续匹配
  - p[0:k-1]、p[i-k:i-1]分别是 p[0:i-1]的最左、最右的，长度为k的真子串

KMP匹配算法

t[j-i]	t[j-i+1]	t[j-i+2]	…	t[j-k]	…	t[j-1]	t[j]
p[0]	p[1]	p[2]	…	p[i-k]	…	p[i-1]	p[i]
	p[0]	p[1]	…		…	p[i-2]	p[i-1]
		p[0]	…		…	p[i-3]	p[i-2]
			…	p[0]	…	p[k-1]	p[k]

逐个测试k=i-1,i-2,i-3,…., 0, -1
直到p[0:k-1]=p[i-k:i-1]且p[k]≠p[i]
第1个通过测试的值用next[i]保存。

如何使用next[i]?

当在p[i]处发生匹配失败，就将字符串p右移使p[next[i]]移到原p[i]所在的位置，即p右移i-next[i]位，再从p[next[i]]开始匹配。

KMP匹配算法

- 性质1：next[i]是一个整数，并且满足

  -1<=next[i]<i
- 性质2：p[i] ≠t[j] -> p[0:i-1]=t[j-i:j-1]

  next[i]的取值应该为使得p[0:i-1]最左、最右的，长度为next[i]的真子串相等
- 性质3：为不丢失可能成功的匹配，当存在多个满足性质2的next[i]时，应取最大的next[i]
- 性质4：p[0:i-1]不存在任意长度最左最右相同的真子串时，next[i]=0。当i=0时，且p[0] ≠t[j],next[i]=-1
- 性质5： p[0:k-1]=p[i-k:i-1]且p[k]=p[i]

  next[i]=next[k]

KMP匹配算法

- $next[0] = -1$;
- 计算$next[i>0]$:

(1) $k(i)=\max\{k=i-1,i-2,\ldots,0,-1 \mid$ 满足$p[0:k-1]=p[i-k:i-1]\}$

(2) if ($p[k(i)] \neq p[i]$) $next[i]=k(i)$; else $next[i]=next[k(i)]$

逐个测试k=i-1,i-2,i-3,…., 0, -1
直到p[0:k-1]=p[i-k:i-1]且p[k]≠p[i]
第1个通过测试的值用next[i]保存。

当在p[i]处发生匹配失败，就将字符串p右移使p[next[i]]移到原p[i]所在的位置，即p右移i-next[i]位，再从p[next[i]]开始匹配。

KMP匹配算法

请计算next[i]

P[i]	a	b	c	a	b	c	a
i	0	1	2	3	4	5	6
next[i]							

next[0]= -1 ;
计算next[i>0] :
(1) k(i)=max{k=i-1,i-2,...,0,-1 | 满足p[0:k-1]=p[i-k:i-1]}
(2) if (p[k(i)]≠p[i]) next[i]=k(i);
else next[i]=next[k(i)]

next[0]= -1 ;
计算next[i>0] :
(1)k(i) ← p[0:i-1]的最左最右、相同的、最长的、真子串的长度
(2)if (p[k(i)] ≠ p[i]) next[i]=k(i);
else next[i]=next[k(i)]

KMP匹配算法

T＝“aaabcaabcabcaa”， P＝“abcabca”

P[i]	a	b	c	a	b	c	a
i	0	1	2	3	4	5	6
next[i]	-1	0	0	-1	0	0	-1

a	a	a	b	c	a	a	b	c	a	b	c	a	a
a	b	c	a	b	c	a							
	a	b	c	a	b	c	a						
		a	b	c	a	b	c	a					
						a	b	c	a	b	c	a	

KMP匹配算法

- 已知：k(0:i-1)，next[0:i-1]，k(0)=-1;
- if (k(i-1)<0||p[k(i-1)]==p[i-1]) k(i)= k(i-1)+1
- else if (next[k(i-1)]<0 || p[next[k(i-1)]]==p[i-1])

k(i) = next[k']+1;

else …

p[0]	…	p[i-1-k']	…	p[i-2]	p[i-1]	p[i]
	…	p[0]	…	p[k'-1]	p[k']	p[k'+1]
	…		…	p[next[k']-1]	p[next[k']]	p[next[k']+1]

KMP匹配算法

确定k(i):

- k(0)=-1;
- 计算k(i), i>0

(1) k' ← k(i-1)
(2) while (k' >=0 && p[k'] != p[i-1]) k' ← next[k'];
(3) k(i) ← k'+1;

KMP匹配算法

- 计算next数组（教材程序的修改）

```
void GenKMPNext(int * next, CMyString* s)
{   next[0]=-1;
    int k=-1;    //k(0)←-1
    int i=1;
    while(i<s->length) //计算next[i]
    {
      //now, k' == k(i-1)
      //while (k' >=0 && p[k'] != p[i-1])  k' ← next[k'];
      while(k>=0&&s->str[k]!=s->str[i-1]) k=next[k];
      k=k+1; //k(i) ← k'+1
      if(s->str[k]!=s->str[i])
                next[i]= k;
      else
                next[i]= next[k];
      i++;
    }
}
```

next[0]= -1;
计算next[i>0]:
(1) k(i) ← p[0:i-1]的最左最右、相同的、最长的、真子串的长度
(2) if (p[k(i)] ≠ p[i]) next[i]=k(i);
else next[i]=next[k(i)]

KMP匹配算法

- KMP算法

```
int Find(CmyString* CS, CmyString* s)
{   int i,j,*next=(int*)malloc(sizeof(int)*s->length);
    GenKMPNext(s->str, s->length, next);
    for(i=0,j=0; i<s->length&&j<CS->length)
    {   if(s->str[i]==CS->str[j])   {i++; j++;}
        else if(next[i]>=0) i=next[i];
             else {i=0; j++;}
    }
    if(i>=s->length) return j-s->length;
    else  return -1;
}
```

BM匹配算法

- KMP算法并不是效率最高的算法，实际采用并不多。各种文本编辑器的"查找"功能（Ctrl+F），大多采用Boyer-Moore算法。

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

BM匹配算法

- 坏字符

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

A B A B C

BM匹配算法

- 为坏字符的情况准备Delta1数组
  - 记录每个字符距离最右边的距离
  - 当坏字符x在第j位匹配出现时，将模式右移Delta1[x]+j-m+1 (m位模式的长度)

*	A	B	A	B	C
5	2	1	2	1	0

BM匹配算法

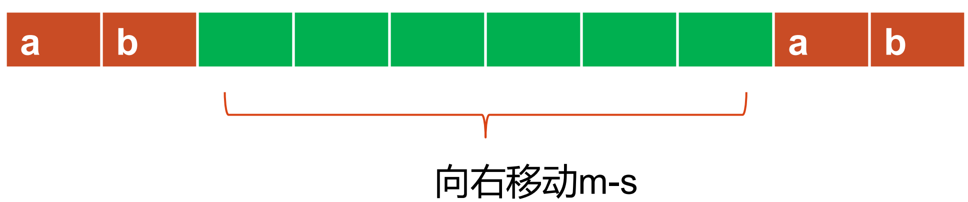
- 好后缀

A A A B C A B A B A B C

A B A B C

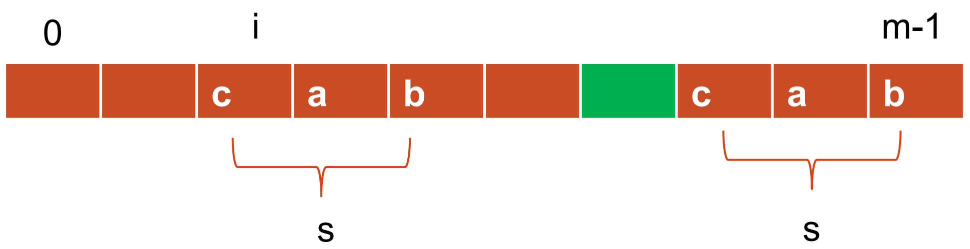
BM匹配算法

- 为好后缀的情况准备Delta2数组
  - 第一种情况：模式串中没有子串匹配上好后缀
    - 向右移动m
  - 第二种情况：模式串中没有子串匹配上好后缀，但找到一个最大前缀



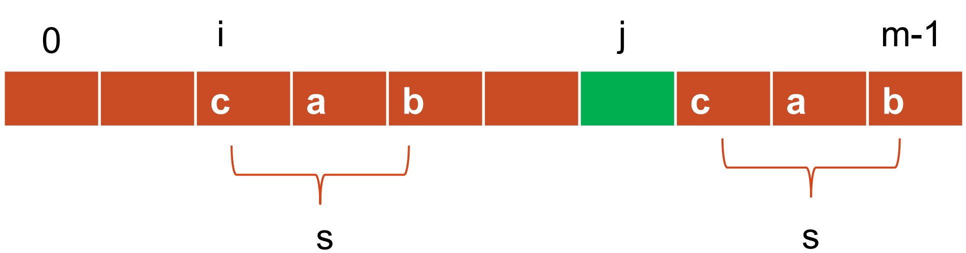
BM匹配算法

- 为好后缀的情况准备Delta2数组
  - 第三种情况：模式串中有子串匹配上好后缀



BM匹配算法

- 数组suffix[]，其中suffix[s] = i 表示好后缀中的某个子串的长度为 s，它在模式串中的前缀子串中有相等的, 且起始位置为 i,如果不存在,我们就记录为 suffix[s] = -1
- prefix 数组，来记录模式串的后缀子串(好后缀的后缀子串)是否能匹配模式串的前缀子串.



BM匹配算法

```
int k = m - j - 1;
if (suffix[k] != -1)
        { return j - suffix[k] + 1; }
for (int i = k - 1; i >= 0; i--)
        { if (prefix[i])
                { return m - i; }
        }
return m;
```

BM匹配算法

- 在匹配到模式串第k位字符为x失败时：
  - 坏字符规则告诉我们，右移Delta1[x]+j-m+1
  - 好后缀规则告诉我们，右移Delta2[k]

右移Max(Delta1[x]+j-m+1, Delta2[k])

BM匹配算法

- acfacf
  - Delta1=

*	a	c	f	a	c	f
6	2	1	0	2	1	0

  - Delta2=

index	1	2	3	4	5	6
suffix	2	1	0	-1	-1	0

index	1	2	3	4	5	6
prefix	F	F	T	F	F	T

	a	c	f	a	c	f
index	0	1	2	3	4	5
Delta2	3	3	3	3	3	6