算法与数据结构

字符串相关算法

setoy@qq.com 2020.6

算法与数据结构

一些基础

1. char：ASCII

- 美国标准信息交换代码
- 8个二进制位，大小为1Byte
- 可以视为整数：
  - 10 - ‘\n’(newline)
  - 13 - ‘\r’(return)
  - 32 - 空格
  - 48 - ’0’
  - 65 - ’A’
  - 97 - ’a’

2. char*和string

DATA STRUCTURE & ALGORITHM

- char a[maxn]; //c字符串要以’\0’结尾
- string b; //c++字符串不规定结尾符，但一般编译器也都已’\0’为结尾符处理
- 两套系统的转换：
  - scanf("%s", a); printf("%s\n", a);
  - cin >> b; cout << b << endl;
  - strcpy(a, b.c_str());
  - b = string(a);

2. char*和string

DATA STRUCTURE & ALGORITHM

char* 的特点	string 的特点
使用函数进行操作	面向对象
不能直接赋值，加减，大小比较	直接用运算符
常数小	常数较大
方便乱搞	操作内部元素受限

- `char a[maxn], b[maxn], c;`
- `memset(a, 0, sizeof(a));`
- `strlen(a);`
- `// combine b after a`
- `strcat(a, b);`
- `strcmp(a, b);`
- `// find the first position of c in a`
- `strchr(a, c);`
- `// find the first position of b in a`
- `strstr(a, b);`

- `string a, b; char c`
- `memset(a, 0, sizeof(a));`
- `a.size(), a.length();`
- `a += b;`
- `a < b ……`
- `a.find(c)`
- `a.find(b)`

例：xoj 3315

DATA STRUCTURE & ALGORITHM

- XOJ 3315，Luogu 3370
- 给出n个字符串，求不同字符串的个数（n最大20000，长度1500，内存限制1M)
- 7
- aaAa
- Aaa
- bbb
- 12345
- bbb
- BbB
- Aaa
- 输出:5

3. 字符串Hash

DATA STRUCTURE & ALGORITHM

- 把字符串映射成一个整数，如abcde映射成12345。但这样会超过int或long long，解决办法是mod p，这里p需要一个素数。
- 因此Hash函数一般是这样设计的：

$$Hash = \sum_{i=0}^{len-1} s[i] \times b^{len-i-1} \bmod p$$

- 其中：s[i]是字符串s的第i位对应的值，可以是ASCII码值，也可以是26个字母的序号值
- len是s的长度
- 可以理解成把字符串序号值看出b进制，再转成10进制的结果就是其Hash值
- b的取值要大于s[i]最大的对应值（从进制转角度理解）

3. 字符串Hash

DATA STRUCTURE & ALGORITHM

- 如s=abc，b取13，p取101，则迭代过程是：
- a → 1 mod p = 1
- ab → (a*13+2) mod p = 15
- abc → (15*13 + 3) mod p = 97

3. 字符串Hash黑科技—自然溢出

- unsigned int 自带对2^{32} 取模.
- unsigned long long 自带对2^{64} 取模.
- 优点：不使用mod(%) 可优化程序的常数.
- 缺点：容易被特殊情况卡掉.

xoj 3315自然溢出

DATA STRUCTURE & ALGORITHM

- b取131，p为2^{64}

```
typedef unsigned long long ull;
ull base=131, a[20010];
char s[20010];
int n, ans=1;

ull hashs(char s[]) {
    int len = strlen(s);
    ull ans = 0;
    for (int i=0; i<len; i++)
        ans = ans*base + (ull)s[i];
    return ans;
}
```

```
int main() {
    scanf("%d", &n);
    for (int i=1; i<=n; i++) {
        scanf("%s", s);
        a[i] = hashs(s);
    }
    sort(a+1, a+n+1);
    for (int i=2; i<=n; i++)
        if (a[i] != a[i-1])
            ans++;
    printf("%d\n",ans);

    return 0;
}
```

xoj 3315双哈希

DATA STRUCTURE & ALGORITHM

- 取两个模数p1和p2
  - 可以用一对孪生素数：1e9+7和1e9+9
  - b的取值规则同单hash
- pair<hash1, hash2>表示一个字符

$$Hash_1 = \sum_{i=0}^{len-1} s[i] \times b^{len-i-1} \bmod p_1$$

$$Hash_2 = \sum_{i=0}^{len-1} s[i] \times b^{len-i-1} \bmod p_2$$

xoj 3315双哈希——自定义结构体

DATA STRUCTURE & ALGORITHM

```
const ull mod1 = 1e9+9, mod2 = 1e9+7;
ull base = 131;
struct data { ull x,y; } a[maxn];
char s[maxn];  int n, ans=1;
ull hashs(char s[], ull modx) {
    int len = strlen(s); ull ans = 0;
    for (int i=0; i<len; i++) ans = (ans*base + (ull)s[i]) % modx;
    return ans;
}
bool comp(data a, data b) { return a.x<b.x; }
int main() {
    scanf("%d",&n);
    for (int i=1; i<=n; i++) {
            scanf("%s",s);  a[i].x = hashs(s, mod1);  a[i].y = hashs(s, mod2);
    }
    sort(a+1, a+n+1, comp);
    for (int i=2; i<=n; i++)  if (a[i].x!=a[i-1].x || a[i-1].y!=a[i].y)  ans++;
    printf("%d\n",ans);
    return 0;
}
```

xoj 3315双哈希——pair

DATA STRUCTURE & ALGORITHM

```
const ull mod1 = 1e9+9, mod2 = 1e9+7;  ull base = 131;
pair<ull, ull> a[maxn];
char s[maxn];  int n, ans = 1;
ull hashs(char s[], ull modx) {
    int len = strlen(s);  ull ans = 0;
    for (int i=0; i<len; i++) ans = (ans*base + (ull)s[i]) % modx;
    return ans;
}
int main() {
    scanf("%d",&n);
    for (int i=1; i<=n; i++) {
        scanf("%s",s);  a[i] = make_pair(hashs(s, mod1), hashs(s, mod2));
    }
    sort(a+1, a+n+1); //先按a.first比较，再按a.second比
    for (int i=2; i<=n; i++) //两者的first和second字段各自相等才相等
        if (a[i]!=a[i-1] || a[i-1]!=a[i]) ans++;
    printf("%d\n",ans);
    return 0;
}
```

3.字符串Hash子串的hash值

DATA STRUCTURE & ALGORITHM

- 已知串a的每个前缀的hash值h_i，怎么求a任一子串a[l, r]的hash值呢?
- 通项是这个：

$$h_i = \sum_{i=0}^{len-1} s[i] \times b^{len-i-1} \bmod p_1$$

- 实际上是递推的，且相当于前缀和： $h_i = (h_{i-1} \times b + s[i]) \bmod p$
- 因此h[l..r]=(h[r]-h[l-1]*b^{r-l+1}) mod p
- 如"de"="abcde"-"abc00"
- =12345-123*10^2
- =45

3.字符串Hash 子串的hash值

DATA STRUCTURE & ALGORITHM

- 已知串X和Y各自的hash，求串XY的hash
- $Hash(XY) = Hash(X)*b^{|B|}+Hash(Y)$
- Hash(“abcde”) = Hash(“abc”)*10^2+Hash(“de”)

例：xoj 1163 Oulipo

DATA STRUCTURE & ALGORITHM

- 给定W和T，求T里面有几个W。
- 1<=|W|<=10^4;
- |W|<=|T|<=10^6;
- 输入:
  - 3
  - BAPC //W
  - BAPC //T
  - AZA //W
  - AZAZAZA //T
  - VERDI
  - AVERDXIVYERDIAN
- 输出:
- 1
- 3
- 0

字符串Hash

- 可以水掉绝大部分字符串题的神器
- 万物皆可哈！

算法与数据结构

Manacher

最长回文串

DATA STRUCTURE & ALGORITHM

- 给定一个字符串，求出其最长回文子串。例如：
  - s="abcd"，最长回文长度为 1;
  - s="ababa"，最长回文长度为 5;
  - s="abccb"，最长回文长度为 4，即bccb
- 暴力：枚举子串的一头一尾，判断子串是否回文，$O(n^3)$
- 根据回文特点优化：枚举对称中心，然后向两边枚举，$O(n^2)$

Manacher

DATA STRUCTURE & ALGORITHM

- 思想：用递推的方式求出以任意位置i为对称中心的最长回文串的长度。
- 对原字符串做调整：
  - 为了变成奇数个字符串，在原串首尾及各字符间各插入一个字符不可能出现的字符：
    - 如s=“ababaccabac”
    - 变成s=“#a#b#a#b#a#c#c#a#b#a#c#”
  - 为了简化代码不越界，忽略0号位置：
    - s=“$#a#b#a#b#a#c#c#a#b#a#c#”

Manacher

DATA STRUCTURE & ALGORITHM

- “回文半径”：回文串中最左或最右位置的字符与其对称轴的距离。
- p[i]表示以 i 为中心的最长回文的半径(L/2+1)

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
s[i]	#	a	#	b	#	a	#	b	#	a	#	c	#	c	#	a	#	b	#	a	#	c	#
p[i]	1	2	1	4	1	6	1	4	1	2	1	2	9	2	1	2	1	6	1	2	1	2	1

- p[i]-1正好是原字符串中最长回文串的长度

Manacher

DATA STRUCTURE & ALGORITHM

- 如何求p[i]?
- mx(max_right)：表示目前回文串向右所能触及的最大（最右）位置。
- pos：mx对称轴所在的位置。
- 当i=14时，如何求p[14]?
- 当i=16时，如何求p[16]?

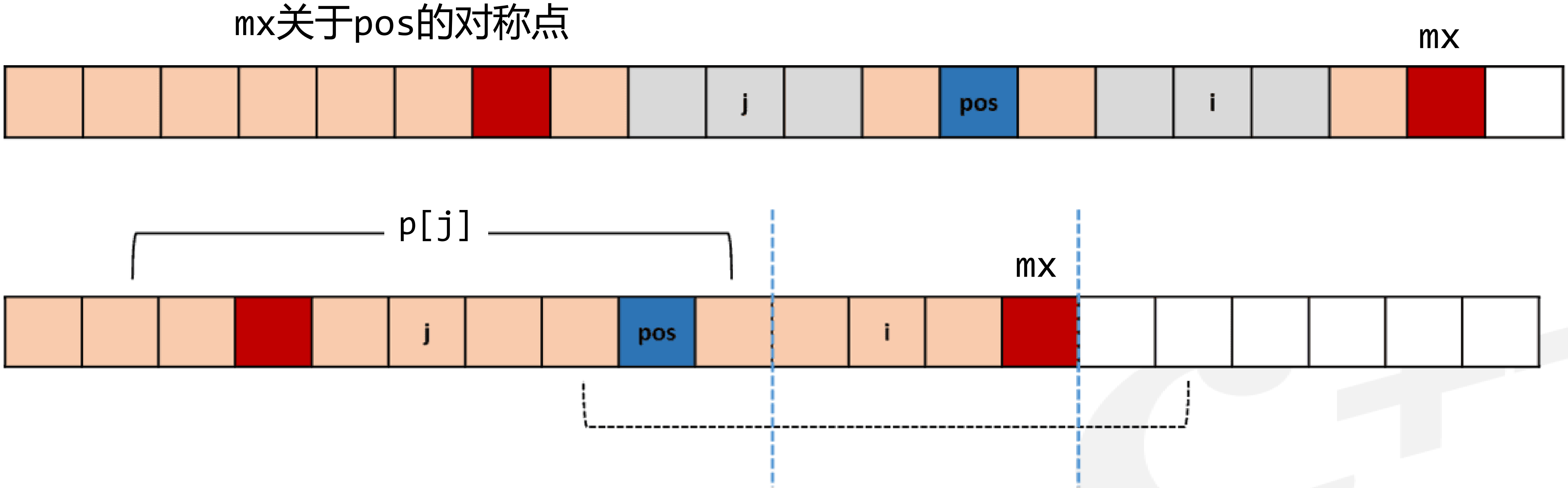
L　　pos　　max_right

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
s[i]	#	b	#	a	#	b	#	c	#	b	#	a	#	b	#	c	#	b	#	a	#	c	#	c	#	b	#	a	#
p[i]	1	2	1	4	1	2	1	8	1	2	1	10	1	?															

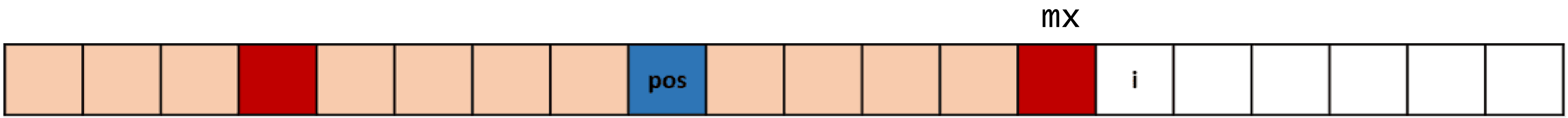
Manacher

DATA STRUCTURE & ALGORITHM

- 当i<mx时
- 由对称性知，p[i]至少可取取p[j]的值，最大可以取到边界mx处
- 再试着以i为对称轴，继续往左右两边扩展，直到左右两边字符不同，或者到达边界。



- 当i>mx时
- 以i为对称轴的回文串还没有任何一个部分被访问过，于是只能从i的左右两边开始尝试扩展了，当左右两边字符不同，或者到达字符串边界时停止



Manacher

DATA STRUCTURE & ALGORITHM

```
char s[maxn], tmps[maxn << 1];
int p[maxn << 1];

int INIT(char *s)
{
    int i, len = strlen(s);
    tmp[0] = '@';
    for(i = 1; i <= 2*len; i += 2)
    {
        tmps[i] = '#';
        tmps[i+1] = s[i/2];
    }
    tmps[2*len+1] = '#';
    tmps[2*len+2] = '$';
    tmps[2*len+3] = 0;
    return 2 * len + 1;
}
```

```
int MANACHER(char *s, int len)
{
        int mx = 0, ans = 0, pos = 0;
        for (int i = 1; i <= len; i++)
        {
            if (mx > i)
                p[i] = min(mx-i, p[2*pos-i]);
            else p[i] = 1;
            while (tmps[i - p[i]] ==
                    tmps[i + p[i]]) p[i]++;
            if(p[i] + i > mx)
            {
                mx = p[i] + i;
                pos = i;
            }
            ans = max(ans, p[i]);
        }
        return ans - 1;
}
```

算法与数据结构

Tier树

XOJ 4728 统计难题

DATA STRUCTURE & ALGORITHM

- Ignatius最近遇到一个难题,老师交给他很多单词(只有小写字母组成,不会有重复的单词出现),现在老师要他统计出以某个字符串为前缀的单词数量(单词本身也是自己的前缀).
- 输入（前部分是单词表，后部分询问该前缀的单词有多少个，所有单词长度不超过10）：
  - banana
  - band
  - bee
  - absolute
  - acm

  - ba
  - b
  - band
  - Abc
- 输出:
  - 2
  - 3
  - 1
  - 0

XOJ 4728 统计难题

DATA STRUCTURE & ALGORITHM

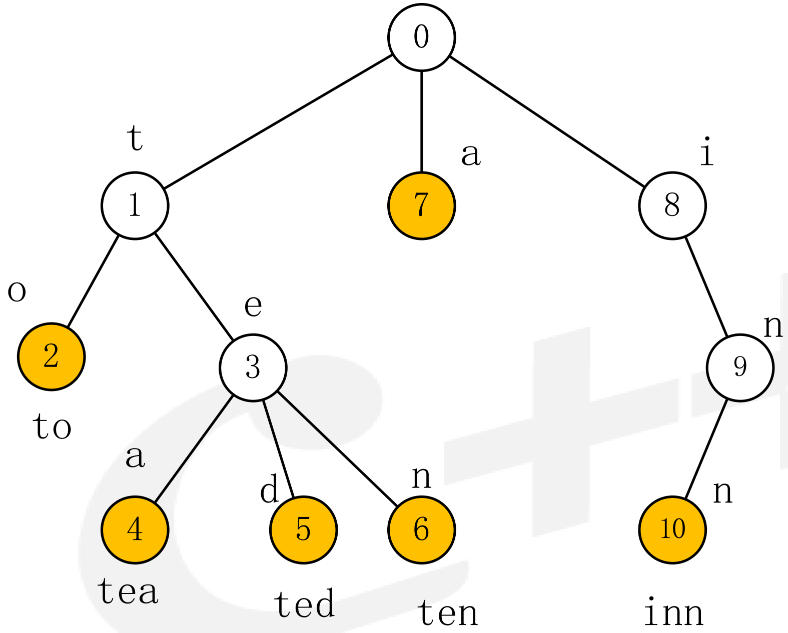
- 算法：Hash、STL map

```
char s[15];
map<string, int> ms;
while (gets(s)) {
    int len = strlen(s);
    if (! len) break;
    for (int i = len; i > 0; i--) {
          s[i] = '\0';
          ms[s]++;
    }
}
while (gets(s))
    printf("%d\n", ms[s]);
```

1.2 Trie树字典树、前缀树

DATA STRUCTURE & ALGORITHM

- Trie树用来保存字符串的集合
  - 根结点为空
  - 内结点一般是字符结点
  - 叶子结点是单词结点
- 用两个数组来保存Trie树：
  - ch[i][j]=x，表示第i号结点的子结点序号是x，保存的字符的ASCII码是j；
  - val[i]，表示第i号结点的附加信息，比如val[i]>0表示i是单词结点。
  - ch[0][19(‘t’)]=1
  - ch[0][0(‘a’)]=7
  - ch[1][4(‘e’)]=3



1.3 Trie树的插入和查询

DATA STRUCTURE & ALGORITHM

- 将Trie树的操作封装到结构体中（OOP风格）：

```
const int maxnode = 400*100+10;
const int sigma=26;
struct Trie
{
  int ch[maxnode][sigma];
  int val[maxnode];
  int sz; //结点总数
  void clear() {
    sz=1; memset(ch[0], 0, sizeof(ch[0]));
  }
  int idx(char c) {return c-'a’;}
  void insert(char *s, int v);
  void find(char *s);
}
```

1.3 Trie树的插入和查询

DATA STRUCTURE & ALGORITHM

- 将Trie树的操作封装到结构体中（OOP的书写风格）：

```
void insert(char *s, int v) //v是附加信息，0表示“非单词结点”
{
    int u=0, n = strlen(s);
    for (int i=0; i<n; i++) {
      int c = idx(s[i]);
      if (!ch[u][c]) {
        memeset(ch[sz], 0, sizeof(ch[sz]));
        val[sz] = 0;
        ch[u][c] = sz++;
      }
      u = ch[u][c];
    }
    val[u] = v;
} //end insert
```

1.3 Trie树的插入和查询

DATA STRUCTURE & ALGORITHM

```
void find(const char *s, int len, vector<int>& ans)
{
    int u=0;
    for (int i=0; i<len; i++) {
      if (s[i] == '\0') break;
      int c = idx(s[i]);
      if (!ch[u][c]) break;
      u = ch[u][c];
      if (val[u] > 0) ans.push_back(val[u]);
    }
  }
} //end struct Trie
```

XOJ 1808 背单词

DATA STRUCTURE & ALGORITHM

- 给出一个由S(最多4000个)个不同单词组成的字典和一个长字符串。把这个字符串按字典分解成若干个单词的连接，有多少种方法？（方法数可能很多，结果对20071027取模)
- 比如有4个单词：a、b、cd、ab，则abcd有两种分解方法：a+b+cd和ab+cd。（字典中单词个数不超过4000)
- 【输入】
- abcd
- 4
- a
- b
- cd
- ab
- 【输出】
- Case 1: 2

XOJ 1808 背单词

DATA STRUCTURE & ALGORITHM

- 用d(i)表示从第i个字符开始的字符串（即后缀s[i..len]）的分解方案数
- 假设在s[i..len]中，仅发现s[i..i+n]是字典中的单词，那么此时d(i)=d(i+n)
- 因此d(i) = sum{d(i+len(x))|x是s[i..len]中的一个前缀字符串，且在字典中出现过}
- 如果枚举x，再判断它是否为s[i..len]的前缀，复杂度是O(4000*len)

XOJ 1808 背单词

DATA STRUCTURE & ALGORITHM

```
char str[30001], tmp[101];
int d[30001];
Trie t;
int main()
{
    while(scanf("%s",str) != EOF){
            scanf("%d",&n);
            t.clear();
            for(int i = 0 ; i < n ; i ++){
                    scanf("%s",tmp);
                    int len = strlen(tmp);
                //将单词的长度保存的trie树节点中
                    t.insert(temp, len);
            }
            printf("Case %d: ",Case++);
            solve();
    }
    return 0;
}
```

XOJ 1808 背单词

DATA STRUCTURE & ALGORITHM

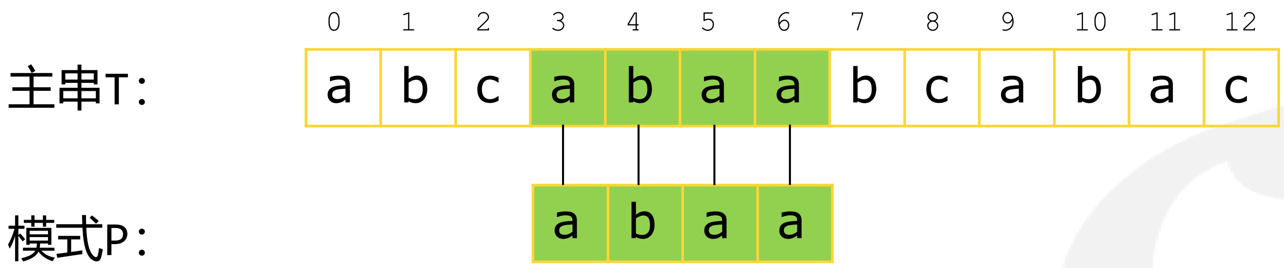
```
void solve()
{
    memset(d,0,sizeof(d));
    int len = strlen(str);
    d[len] = 1;
    for(int i = len - 1; i >= 0 ; i--){
        t.find(str+i,len-i);
        for(int j = 0 ; j < ans.size(); j++){
            d[i] = (d[i]+d[i+ans[j]])%MOD;
        }
    }
    printf("%d\n",d[0]);
}
```

算法与数据结构

KMP

2.1 字符匹配问题

- 在计算机所处理的各类数据中，有很大一类属于正文数据，也常称为文本型数据，如各种文稿资料、源程序、上网浏览页面的HTML文件等。几乎在所有对正文进行编辑的软件中，都提供有“**查找**”的功能，即要求在正文串中，查询有没有和一个“给定的串”相同的子串，若存在，则屏幕上的光标移动到这个子串的起始位置（或高亮显示所有子串）。这个操作即为串的定位操作，通常称为正文模式匹配，即字符串的匹配。
- 子串：某一个主串中任意个连续字符组成的子序列。
  - 主串T=abcabaabcabac，则a、ab、abc、aabca、cabaabca都是它的子串；而aaa、abcd、abbbb都不是其子串。
- 字符串匹配问题：确定一个字符串P（通常称之为模式）是不是另外一个字符串T（即主串）的子串。



2.1 字符匹配问题

- 假设主串T的长度是n，模式P的长度是m，且m≤n
- 字符串匹配的一般问题是，找出主串中所有匹配点i，满足：

  T[i]=P[0], T[i+1]=P[1], ..., T[m-1]=P[m-1]

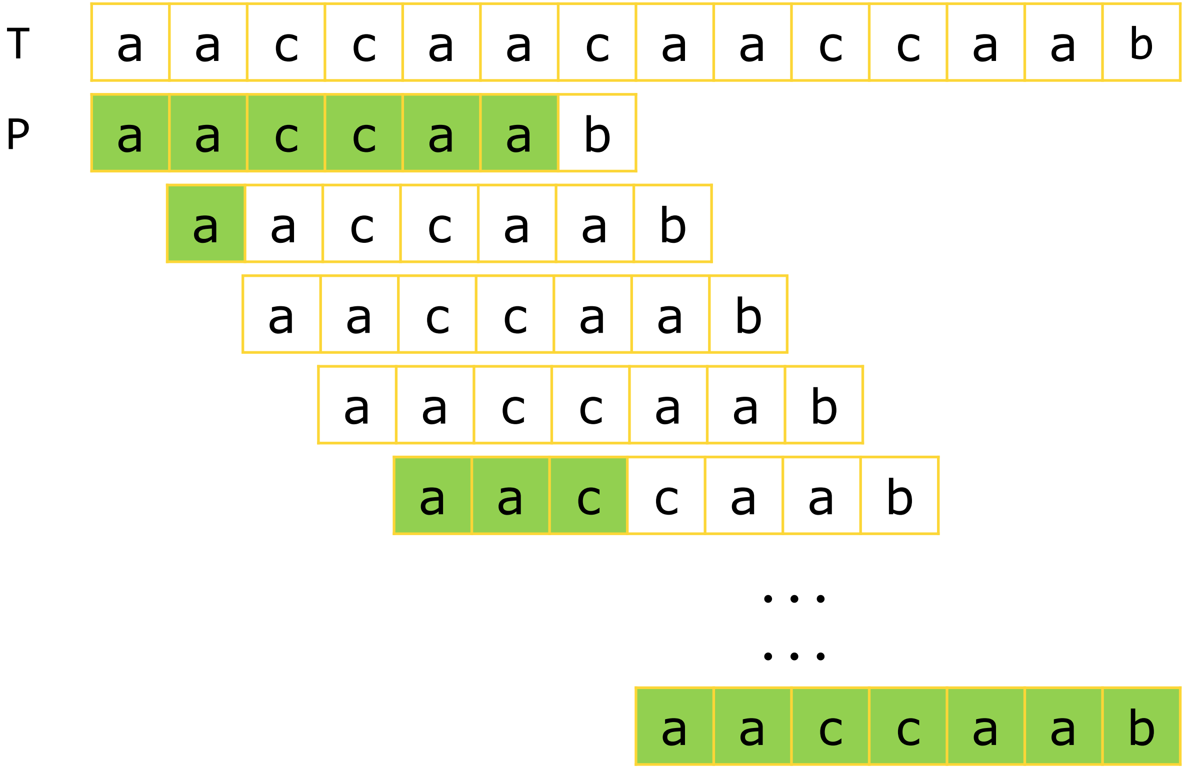
  的所有非负整数i
- 如下图，有两个匹配点i=3、6

	0	1	2	3	4	5	6	7	8	9	10	11	12
主串T:	a	b	c	a	b	a	a	b	a	a	b	a	c
模式P:				a	b	a	a						

2.2 暴力算法

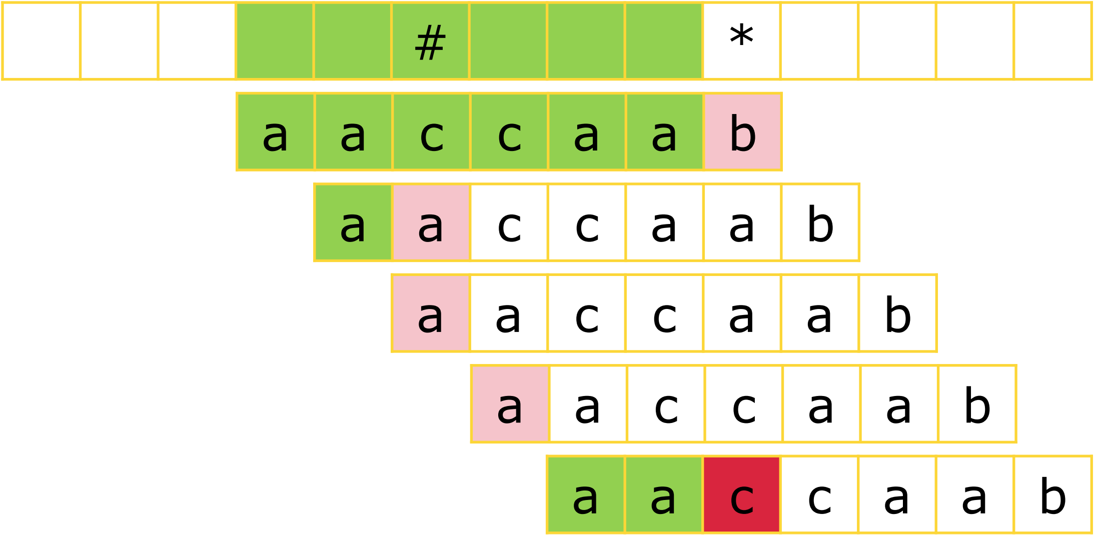
DATA STRUCTURE & ALGORITHM

- 依次判断T的每个位置是不是匹配点，可能的枚举点有n-m个，因此最坏情况下复杂度是O(m(n-m))



2.3 暴力算法的问题

- 假设在比较主串T的*号位置字符时和模式P失配，暴力算法会右移一位、两位...来继续比较，但实际上我们知道绿色部分就是aaccaa，#号位置字符已经比过一次，无需再比，也即右移两位到#号位置也肯定失配，这是有P串本身决定的。而右移4位是有可能匹配的，这时候需要比较*号位置字符和P的第三个字符。那么到底右移几位才合适呢？



2.3 部分匹配时的特征

DATA STRUCTURE & ALGORITHM

- 当t[i]和p[j]失配时，暴力算法会i-j+2，j=0。实际上只要重新调整j，i不变继续比较即可。如何重新调整j呢？显然j越大，算法效率越高。
- 假设j最大可以重新调整到k，这时必须满足p[0..k-1]=T[i-k..i-1]，而T[i-k..i-1]=p[j-k..j-1]，也即p[0..k-1]=p[j-k..j-1]。因此k的取值，必须使得p的前k个字符与“从当前失配位置的前一个字符”开始的k个字符相等。

主串T: a b a b a a b a b a b

模式P: a b a b a b

T[2..4]=P[2..4]

P[0..2]=P[2..4]

主串T: a b a b a a b a b a b

模式P: a b a b a b

T[2..4]=P[0..2]

```
i= 0 1 2 3 4 5 6 7 8 9 10
T= a b c a a a b c d a
P= a b c a b
```

```
i= 0 1 2 3 4 5 6 7 8 9 10
T= a b c a a a b c d a
P=           a b c a b
```

T[3..3]=P[3..3]

T[3..3]=P[0..0]

P[0..0]=P[3..3]

2.3.2 模式串的自匹配

- 求前缀、后缀的最长公共元素的长度
- 判断P=ababacb的前缀和后缀是否相等

a b a b a c b

前缀 后缀

长度	子串	前缀	后缀	最大长度
1	a	-	-	0
2	ab	a	b	0
3	aba	a,ab	a,ba	1
4	abab	a,ab,aba	b,ab,bab	2
5	ababa	a,ab,aba,abab	a,ba,aba,baba	3
6	ababac	a,ab,aba,abab,ababa	c,ac,bac,abac,babac	0
7	ababacb	…	…	0

2.4 验证一般性结论

DATA STRUCTURE & ALGORITHM

- 当T[0..i-1]=p[0..j-1]匹配，而T[i]≠P[j]失配时，j取k，继续比较T[i]和P[k]:

$$k=\begin{cases}0 & (j=1)\\ \max\{k \mid (1\le k\le j)\wedge(P[0..k-1]=P[j-k..j-1])\} & (j>1)\end{cases}$$

```
i= 0 1 2 3 4 5 6 7 8 9 10 11 12 13
T= a b a b a b a a b a b a c b
P= a b a b a c b
j= 0 1 2 3 4 5 6
```

j取3，继续比较T[5]和P[3]

```
i= 0 1 2 3 4 5 6 7 8 9 10 11 12 13
T= a b a b a b a a b a b a c b
P=     a b a b a c b
j=     0 1 2 3 4 5 6
```

j取3，继续比较T[7]和P[3]

```
i= 0 1 2 3 4 5 6 7 8 9 10 11 12 13
T= a b a b a b a a b a b a c b
P=         a b a b a c b
j=         0 1 2 3 4 5 6
```

j取1，继续比较T[7]和P[1]

```
i= 0 1 2 3 4 5 6 7 8 9 10 11 12 13
T= a b a b a b a a b a b a c b
P=             a b a b a c b
j=             0 1 2 3 4 5 6
```

j取0，继续比较T[7]和P[0]

长度	子串	相等前缀	相等后缀	最大长度
1	a	-	-	0
2	ab	a	b	0
3	aba	a	a	1
4	abab	ab	ab	2
5	ababa	aba	aba	3
6	ababac	-	-	0
7	ababacb	-	-	0

```
i= 0 1 2 3 4 5 6 7 8 9 10 11 12 13
T= a b a b a b a a b a b a c b
P=               a b a b a c b
j=               0 1 2 3 4 5 6
```

2.5 KMP算法

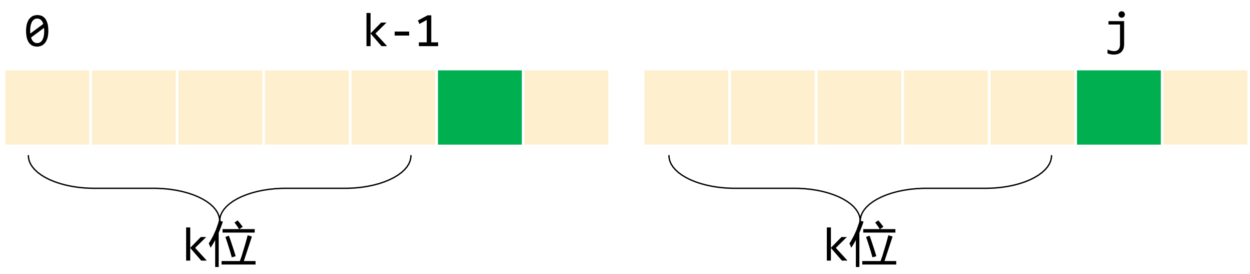
- Knuth-Morris-Pratt算法
- 考察模式P，建立状态机：编号为i的结点表示已经匹配了i个字符（或者说正在匹配第i号字符），匹配开始时的状态是0，成功匹配时状态加1（表示多匹配了一个字符），而失配时，重新调整P的位置。
- 为方便起见，这里用失配函数（Failure Function）F[i]表示i失配时应该调整到的新值，初始F[0]=-1，F[1]=0

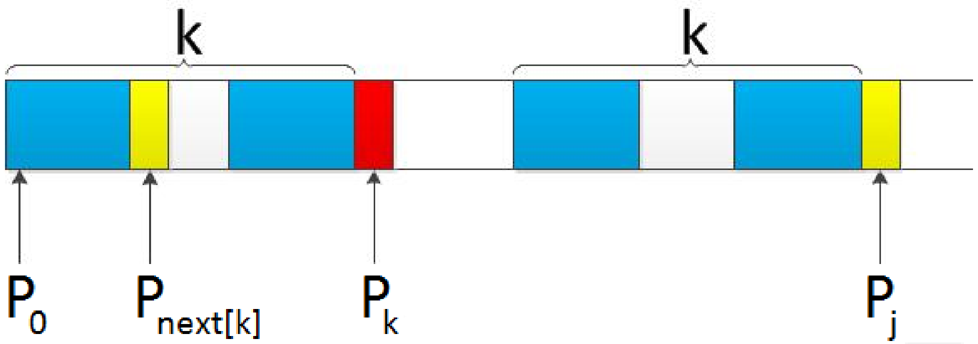
长度	子串	相等前缀	相等后缀	最大长度
1	a	-	-	0
2	ab	a	b	0
3	aba	a	a	1
4	abab	ab	ab	2
5	ababa	aba	aba	3
6	ababac	-	-	0
7	ababacb	-	-	0

	0	1	2	3	4	5	6	7
P	a	b	a	b	a	c	b	
F	-1	0	0	1	2	3	0	0

2.6 失配函数

- 失配函数（Failure Function）F[i]表示i失配时应该调整到的新值，初始F[0]=-1，F[1]=0，如何求F[i]?
- 假设已经求得F[j]=k，如何求F[j+1]?
  - 若P[k]=P[j]，则P[0..k]=P[j-k..j]，即F[j+1]=F[j]+1
  - 若P[k]≠P[j]，相当于当前已失配，如何寻找下一个j呢？可以让新的j取F[j]本身，再看P[F[F[j]]]与P[j]的情况，这就是自我匹配过程，如下表F[4]、F[8]的计算过程





下标	0	1	2	3	4	5	6	7	8
模式串P	a	b	a	a	b	a	a	a	a
F[i]	-1	0	0	1	1	2	3	4	1

2.6 失配函数

DATA STRUCTURE & ALGORITHM

```
get_next(P, F)
{
  F[0] = -1;
  m = P.length - 1;
  k = -1;
  for j = 0 to m
  {
    while (k != -1 && P[j]!=P[k])
      k = F[k];
    F[++j] = ++k;
  }
```

下标	0	1	2	3	4	5	6
模式串P	a	b	a	b	a	c	b
F[i]	-1	0	0	1	2	3	0

```
i= 0 1 2 3 4 5 6 7 8 9 10 11 12 13
T= c a c a b a b a b a b a c b
P= a b a b a c b
j= 0 1 2 3 4 5 6

i= 0 1 2 3 4 5 6 7 8 9 10 11 12 13
T= c a c a b a b a b a b a c b
P=   a b a b a c b
j=   0 1 2 3 4 5 6

i= 0 1 2 3 4 5 6 7 8 9 10 11 12 13
T= c a c a b a b a b a b a c b
P=     a b a b a c b
j=     0 1 2 3 4 5 6

i= 0 1 2 3 4 5 6 7 8 9 10 11 12 13
T= c a c a b a b a b a b a c b
P=       a b a b a c b
j=       0 1 2 3 4 5 6

i= 0 1 2 3 4 5 6 7 8 9 10 11 12 13
T= c a c a b a b a b a b a c b
P=           a b a b a c b
j=           0 1 2 3 4 5 6
```

2.5 KMP算法

DATA STRUCTURE & ALGORITHM

```
KMP-Matcher(T, P)
{
  n = T.length;
  m = P.length;
  get_next(P, F);
  j = 0;
  for i = 0 to n-1 {
    while (j != -1 && P[j] != T[i])
      j = F[j];
    i++, j++;
    if (j >= m) {
      cout << i - m + 1; //找到匹配点
      j = F[j];
    }
  }
}
```

i=	0	1	2	3	4	5	6	7	8	9	10	11	12	13
T=	c	a	c	a	b	a	b	a	b	a	b	a	c	b
P=						a	b	a	b	a	c	b		
j=						0	1	2	3	4	5	6		

2.7 失配函数的不足之处

DATA STRUCTURE & ALGORITHM

- KMP在这种情况下，显得很笨拙

```
i= 0 1 2 3 4 5 6 7 8
T= a a a b a a a a b
P= a a a a b
j= 0 1 2 3 4 5 6
```

```
i= 0 1 2 3 4 5 6 7 8
T= a a a b a a a a b
P=   a a a a b
j=   0 1 2 3 4 5 6
```

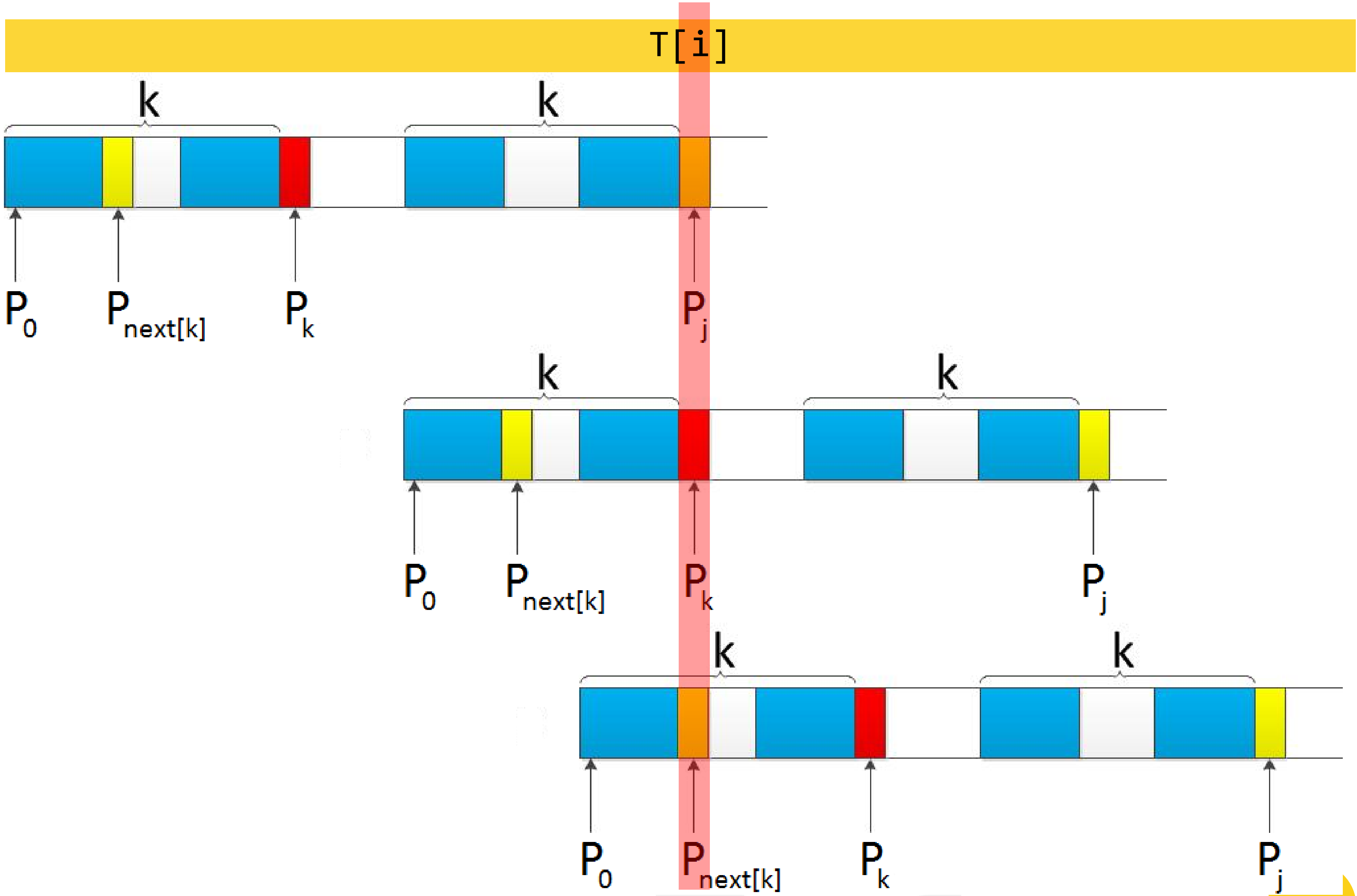
```
i= 0 1 2 3 4 5 6 7 8
T= a a a b a a a a b
P=     a a a a b
j=     0 1 2 3 4 5 6
```

下标	0	1	2	3	4
模式串P	a	a	a	a	b
F[i]	-1	0	1	2	3

2.7.2 失配函数的优化思路

DATA STRUCTURE & ALGORITHM

- 若T[i]!=P[j]，则模式串P滑动到F[j]（图示P[k]位置）
- 若T[i]!=P[k]，则模式串再滑动到P[next[k]]
- 因此，如果P[j]==P[k]，那么当i位失配时，可以直接滑动到P[next[k]]位置，减少滑动次数



2.7.3 优化后的失配函数

DATA STRUCTURE & ALGORITHM

```
get_next(P, F)
{
  F[0] = -1;
  m = P.length - 1;
  k = -1;
  for j = 0 to m
  {
    while (k != -1 && P[j]!=P[k])
        k = F[k];
    if (P[++j] == P[++k])
        F[j] = F[k]
    else
        F[j] = k
  }
```

```
i= 0 1 2 3 4 5 6 7 8
T= a a a b a a a a b
P= a a a a b
j= 0 1 2 3 4 5 6
```

```
i= 0 1 2 3 4 5 6 7 8
T= a a a b a a a a b
P=         a a a a b
j=         0 1 2 3 4
```

下标	0	1	2	3	4
模式串P	a	a	a	a	b
F[i]	-1	0	-1	-1	3

算法与数据结构

最小表示

Alfred Aho, Margaret Corasick

最小表示

DATA STRUCTURE & ALGORITHM

- 字符串S[1..n]，如果不断把它最后一个字符放到开头，就会得到n个字符串，这n个字符串是循环同构的（下面的S、T循环同构）
- S[i..n]+S[1..i-1] = T
- 如S="abca"，它的4个循环同构字符串是"abca","aabc","caab","bcaa"
- 字典序最小的就是S的最小表示

最小表示

- 考虑S="bacacabc"，如何得到最小表示，即如何得到开始段最小?
  - 最小段是"abc"，只需将"bacac"移到它后面即可
- 初始化指针i=0，j=1
- 当S[i]>S[j]时，i++，且i==j时，i++
- 当S[i]<S[j]时，j++，且i==j时，i++
- 当S[i]==S[j]时，说明当前位置有相同元素，取谁都可以
  - 因此需要一个变量保存相同元素的长度k
- 于是各个位置用(i+k)%n来表示，算法描述为：
  - 当s[(i+k)%n]==s[(j+k)%n]时，k++
  - 当s[(i+k)%n]>s[(j+k)%n]时，i+=k+1，且i==j时，i++
  - 当s[(i+k)%n]<s[(j+k)%n]时，j+=k+1，且i==j时，i++
  - 最后取min(i,j)的位置

最小表示


```
while (i < n && j < n) {
    for (k = 0; k < n && s[(i+k) % n] == s[(j+k) % n]; k++);
    if (k == n) break;
    if (s[(i+k) % n] > s[(j+k) % n]) {
         i += k+1;
         if (i == j) i++;
    } else {
         j += k+1;
         if (i == j) j++;
    }
}
int ans = min(i,j)
```

算法与数据结构

Aho-Corasick自动机

Alfred Aho, Margaret Corasick

XOJ 4729 Keywords Search

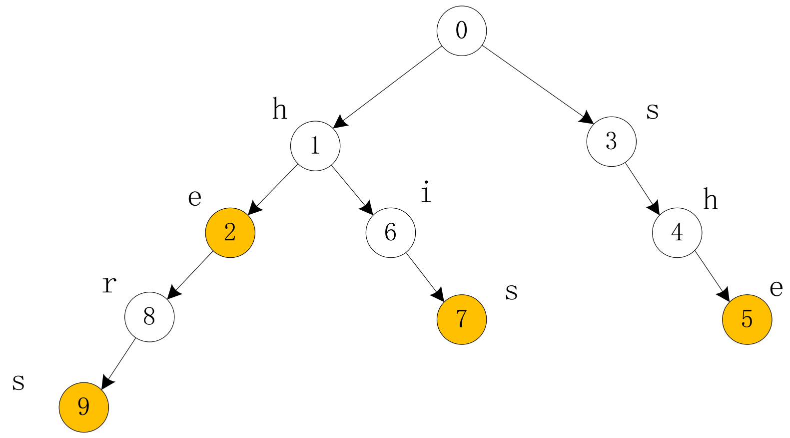
DATA STRUCTURE & ALGORITHM

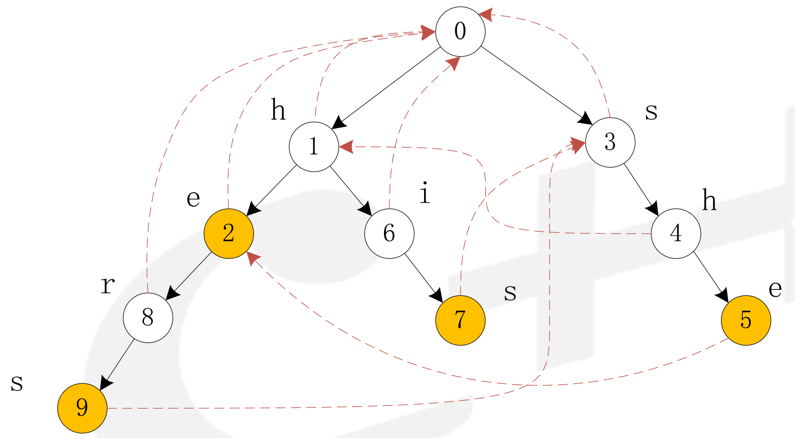
- 有多个关键字，请在一个文本中找到它们。
- 输入：
  - 5 //5个模式字符串
  - she
  - he
  - say
  - shr
  - her
  - yasherhs //目标字符串T
- 请统计有几个模式串在T中出现了，样例输出是3

1. AC自动机

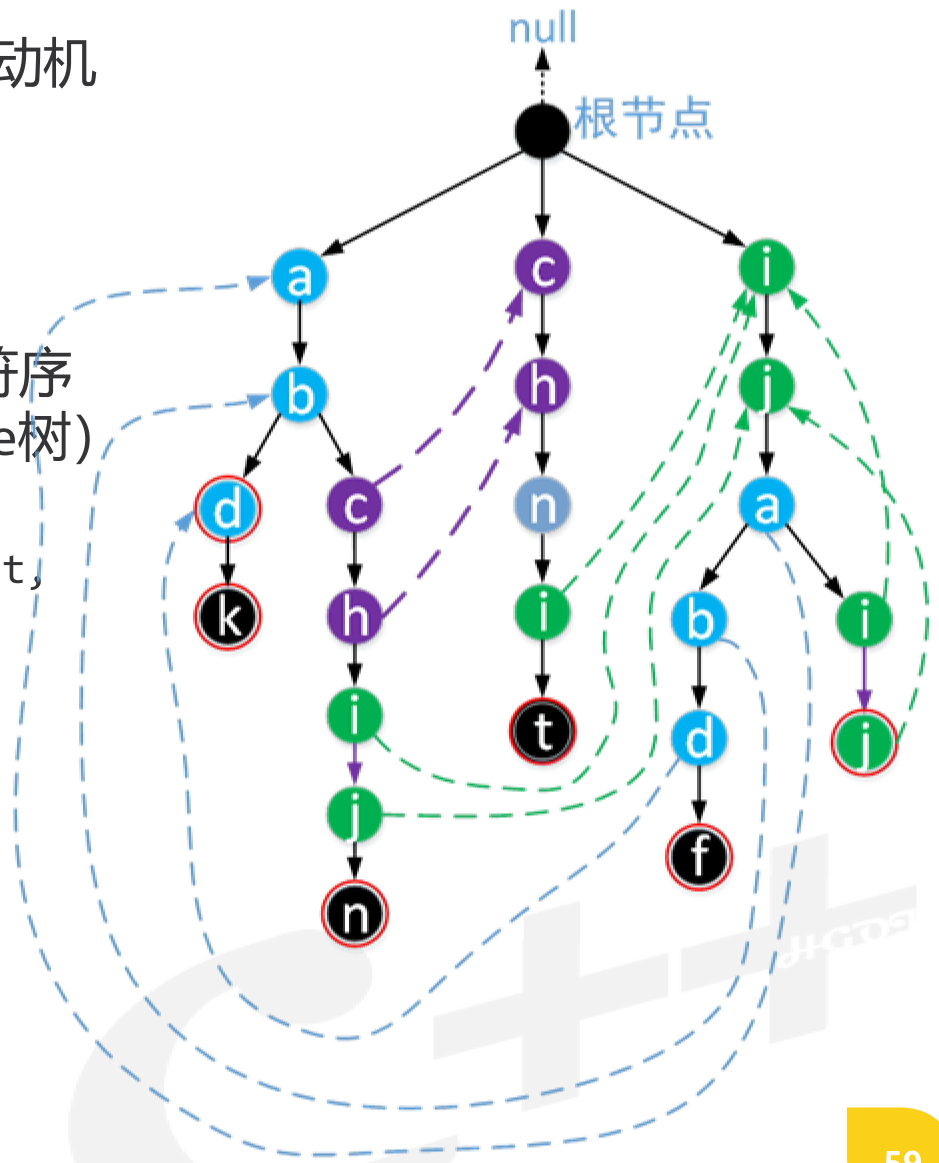
DATA STRUCTURE & ALGORITHM

- KMP：单模式匹配
- AC自动机：多模式匹配
  - 左图：模式{he,she,his,hers}的Trie树
  - 右图：Trie树对应的AC自动机
- 若要找hershe的匹配，可以经过1、2、8、9匹配一个模式串hers，然后再从根节点开始重新匹配吗？
- 可以从3开始
- 这就需要建立失配数组





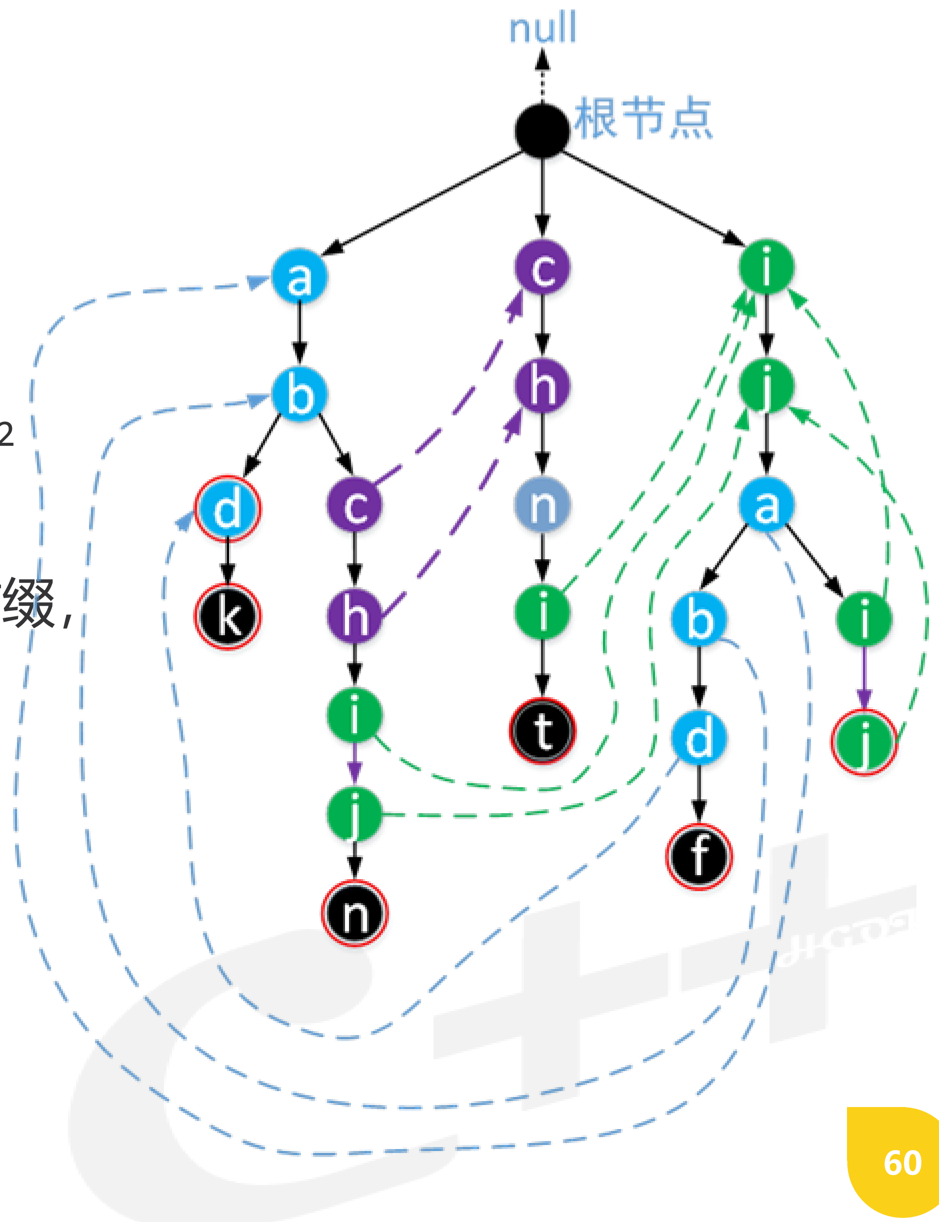
- {abd,abdk, abchijn, chnit, ijabdf, ijaij}的AC自动机
  - 根结点不存储任何字符，根结点的fail指针为null
  - 虚线表示该结点的fail指针的指向
  - 字符串的最后一个字符的结点外部都用红圈表示
  - 所有指向根结点的fail虚线都未画出
- 每个结点的fail指针表示：由根结点到该结点所组成的字符序列的所有后缀 和 整个目标字符串集合（也就是整个Trie树）中的所有前缀 两者中最长公共的部分。
  - 如ijabdf中以“d”结尾的所有后缀，在{abd,abdk, abchijn, chnit, ijabdf, ijaij}所有前缀中最长公共部分就是abd



1.3 AC自动机匹配（查找）过程

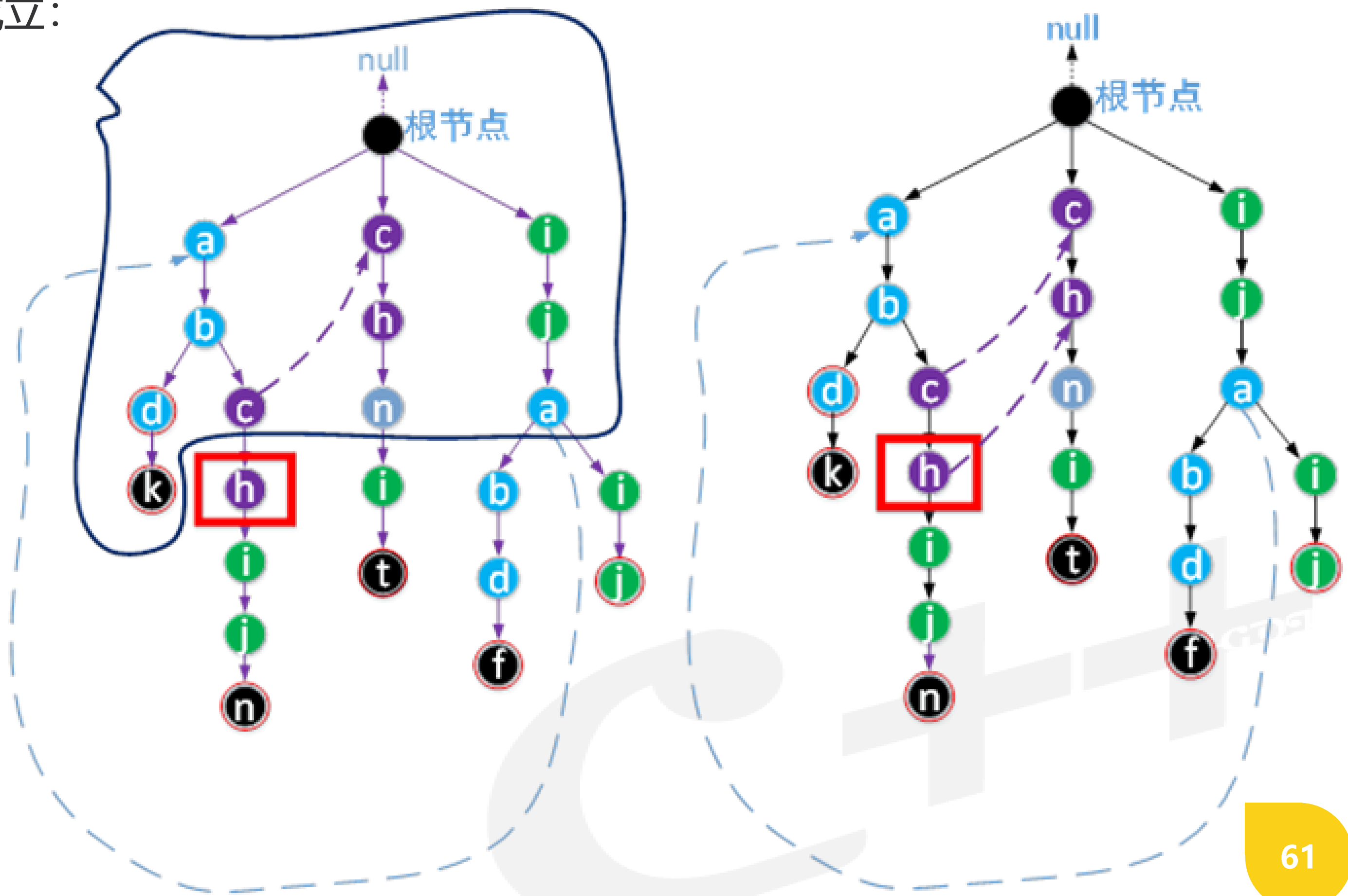
DATA STRUCTURE & ALGORITHM

- T=“abchnijabdfk”
- 1. j=0，指向根节点
- 2. 读入T的下一个字符
- 3. 从当前节点的所有子节点中寻找匹配点：
  - 若成功：
    - 判断当前节点以及fail指向的节点是否表示一个字符串的结束
    - 若是，则记录该子串的索引起点位置（当前索引-字符串长度+1），继续执行2
  - 若失败：
    - 执行4
- 4. 若fail==null，说明Trie树中没有任何字符串是T的前缀，重启状态机（j=0），执行2；
- 5. 否则当前j指向fail节点，执行3



- 先将所有目标字符串插入到Trie树中，然后BFS为每个结点的所有孩子节点的fail指针找到正确的指向
- 1. 根节点的所有孩子的fail都指向根，然后根的所有孩子入队
- 2. 当队列不空时：
  - 2.1 队首出队，记为fa，fa的失配指针记为v = F[fa]
  - 2.2 判断fa.child[i] == v.child[i]是否成立：
    - 若成立： F[fa.child[i]] = v.child[i]
    - 不成立： v = F[v]，继续执行2.2
  - 2.3 fa.child[i]入队，执行2

- 如右图1的c字符已经建立好了F[]数组，如何确定h的F[]失配指针呢?
- 结果如右图2所示

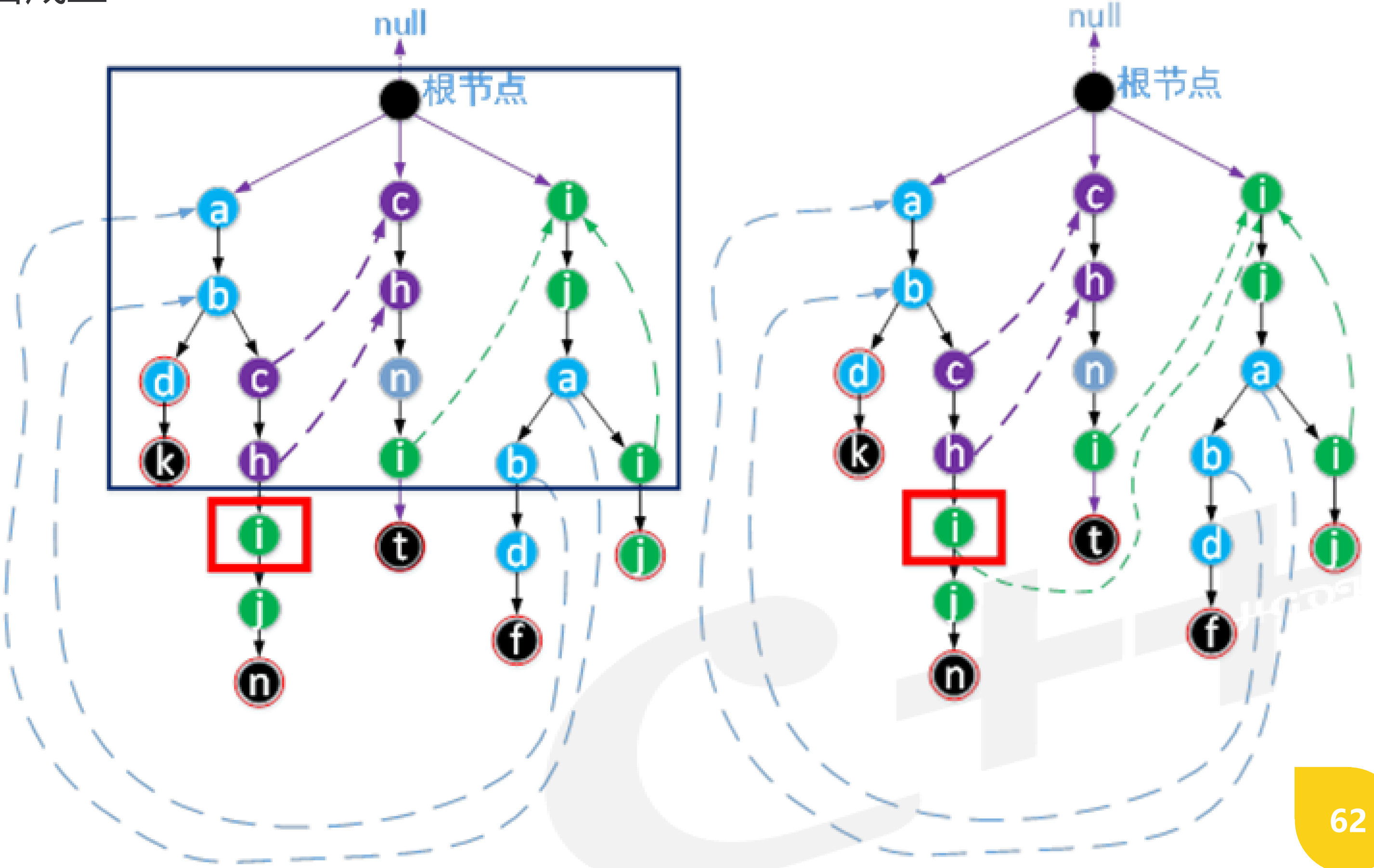


1.4 失配函数（AC自动机的构造）（样例2）

DATA STRUCTURE & ALGORITHM

- 先将所有目标字符串插入到Trie树中，然后BFS为每个结点的所有孩子节点的fail指针找到正确的指向
- 1. 根节点的所有孩子的fail都指向根，然后根的所有孩子入队
- 2. 当队列不空时：
  - 2.1 队首出队，记为fa，fa的失配指针记为v = F[fa]
  - 2.2 判断fa.child[i] == v.child[i]是否成立：
    - 若成立： F[fa.child[i]] = v.child[i]
    - 不成立： v = F[v]，继续执行2.2
  - 2.3 fa.child[i]入队，执行2

- 如右图1的h字符已经建立好了F[]数组，如何确定i的F[]失配指针呢？
- 结果如右图2所示



1. AC自动机

DATA STRUCTURE & ALGORITHM

```
struct aho_corasick {
    int ch[maxn][sigma];
    int F[maxn];
    int val[maxn], last[maxn];
    bool vis[maxn];
    int sz;
    aho_corasick() {
        sz = 1; memset(ch[0], 0, sizeof(ch[0])); memset(vis, 0, sizeof(vis));
    }
    int idx(char c) {
        return c - 'a';
    }
    // 插入字符串，Trie建树
    void insert(char *s) {}
    // 打印
    void print(int j) { if (j && !vis[j]) ans += val[j], vis[j] = true, print(last[j]); }
    // 在T中找模式串
    int find(char *T) {}
    // 失配函数
    void get_fail() {}
};
```

2. 插入字符串，Trie建树


```
void insert(char *s) {
    int u = 0, n = strlen(s);
    for (int i = 0; i < n; i++) {
        int c = idx(s[i]);
        if (! ch[u][c]) {
            memset(ch[sz], 0, sizeof(ch[sz]));
            val[sz] = 0;
            ch[u][c] = sz++;
        }
        u = ch[u][c];
    }
    val[u]++;
}
```

3. 在T中找模式串


```
int find(char *T) {
    int n = strlen(T);
    int j = 0; // 当前节点编号，初始为root
    for (int i = 0; i < n; i++) {
        int c = idx(T[i]);
        //while (j && !ch[j][c]) j = F[j];
        j = ch[j][c];
        if (val[j])
            print(j);
        else if (last[j])
            print(last[j]);
    }
}
```

4. 失配函数

DATA STRUCTURE & ALGORITHM

```
void get_fail() {
    queue<int> q; F[0] = 0;
    for (int c = 0; c < sigma; c++) {
        int u = ch[0][c];
        if (u) { F[u] = 0,  q.push(u), last[u] = 0; }
    }
    while (! q.empty()) {
        int fa = q.front(); q.pop();
        for (int c = 0; c < sigma; c++) {
            int u = ch[fa][c], v = F[fa];
            if (!u) { ch[fa][c] = ch[v][c]; continue; }
            q.push(u);
            while (v && !ch[v][c]) v = F[v];
            F[u] = ch[v][c];
            last[u] = val[F[u]] ? F[u] : last[F[u]];
        }
    }
}
```

算法与数据结构

后缀数组

Suffix Array, Suffix Tree

1. 后缀树&后缀数组

DATA STRUCTURE & ALGORITHM

- 能干啥?
- 高效解决绝大部分字符串问题:
  - 查找子串（KMP能做）
  - 最长重复子串
  - 最长公共子串（DP能做）

1.1 后缀数组

DATA STRUCTURE & ALGORITHM

- 后缀：从某个位置i开始，到整个串末尾的子串，用Suffix(i)表示
- 如字符串 "banana" 的所有后缀：
  - Suffix(0) = "banana"
  - Suffix(1) = "anana"
  - Suffix(2) = "nana"
  - Suffix(3) = "ana"
  - Suffix(4) = "na"
  - Suffix(5) = "a"

- 后缀数组（SA）：将所有后缀从小到大排序，将排好序的后缀下标i存入数组：
  - Suffix(5) = "a"
  - Suffix(3) = "ana"
  - Suffix(1) = "anana"
  - Suffix(0) = "banana"
  - Suffix(4) = "na"
  - Suffix(2) = "nana"
- SA[] = {5,3,1,0,4,2}

- 名次数组：Suffix(i)在后缀数组中的排名（第1名开始）
  - Suffix(0) = “banana” ▪ 第4名，rank[0] = 4
  - Suffix(1) = “anana” ▪ 第3名，rank[1] = 3
  - Suffix(2) = “nana” ▪ 第6名，rank[2] = 6
  - Suffix(3) = “ana” ▪ 第2名，rank[3] = 2
  - Suffix(4) = “na” ▪ 第5名，rank[4] = 5
  - Suffix(5) = “a” ▪ 第1名，rank[5] = 1
- SA[] = {5,3,1,0,4,2}
- rank[] = {4,3,6,2,5,1}
- 由rank[3] = 2，可知Suffix(3)排名第2，也即SA[2-1]必然等于3
- 反之，由SA[5]=2，可知Suffix(2)排名第5+1，也即rank[2]必然等于6

rank[3] = 2
下标i 也即后缀i　名次

SA[5] = 2
名次　下标i 也即后缀i

2. 后缀数组的构造

DATA STRUCTURE & ALGORITHM

- DC3 (Difference Cover modulo 3) VS 倍增算法
- 对每个下标开始的长度为2^k的子串进行排序，求出rank值
- $k = 0,1,2,\cdots$，当$2^k \geq n$时结束
- 每次排序利用上一次的rank结果

	DC3	倍增
时间复杂度	$O(n)$卡常	$O(nlogn)$
空间复杂度	$O(n)$	$O(n)$
编程复杂度	高	较低

后续学习

DATA STRUCTURE & ALGORITHM

- 后缀自动机（SAM）
- SAM+线段树

练习

DATA STRUCTURE & ALGORITHM

- XOJ 1808 Remember the word
- XOJ 4246 IMMEDIATE DECODABILITY
- XOJ 3948 Phone list
- XOJ 1163, POJ 3461 Oulipo
- XOJ 1160, POJ 2406 Power Strings
- XOJ 1162, POJ 2752 Seek the Name, Seek the Fame
- XOJ 1161, POJ 1961 Period