



# 字符串

2018.8.7



# KMP & 擴展KMP

2018-08-06

# 模式串与文本串

模式串

PATTERN:

"CO"

文本串

TEXT:

POSITION:

1 2 3 4 5 6 7 8  
"COACOLA"



# 两层for循环的匹配: $O(n*m)$

coca

cola


CO

CO

.....

## NAIVE SHIFTING ALGORITHM

```
n ← length|Text|
m ← length|Pattern|
for pos ← 0 to n-m do
    j ← 0
    while j < m and T[pos+j] = Pattern[j] do
        j ← j+1
    if j ≥ m then
        return pos
return "no valid pos"
```



**$O(NM)$**

# 当文本串很长的时候

TEXT: IMTRYINGTOMATCHAPATTERNSOMEWHEREIN  
CO

THISLONGPIECEOFTEXTOWELLIGUESSITSN

OTHERENOPATTERNMATCHINGTODAYIMGOIN

GTOGOHOMERELAXANDHAVESOMECOCOPOPS

# 问题出在哪儿

模式串每次只向后移动一位，导致重复计算。

WXYZJSFDGSDFWFVEB

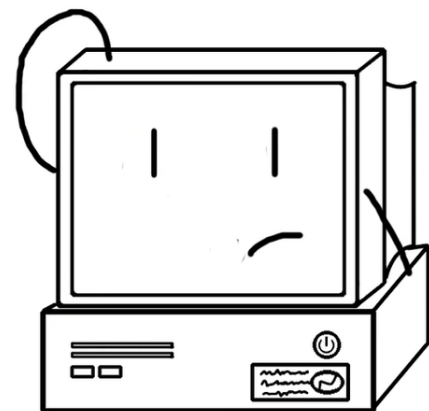
ABCD

ABCD

ABCD

明显地看出前几次匹配是没有用的。然而还是只能每次移动一位。

# PROBLEM



一个更快的方案：  $O(n+m)$



The Knuth-Morris-Pratt (KMP) String Matching Algorithm

# KMP的构成

get\_next(): 预处理出**模式串T**的Next数组

KMP(): 字符串匹配



# Next数组

含义：模式串T前i个字符的【前缀】和【后缀】的最大公共长度。

例：T = "ABC AAABDC ABCAB"

T的前i个字符		Next[i]			
i=1	A	0	i=8	ABC AAABD	0
i=2	AB	0	i=9	ABC AAABDC	0
i=3	ABC	0	i=10	ABC AAABDC A	1
i=4	ABC A	1	i=11	ABC AAABDC AB	2
i=5	ABC AA	1	i=12	ABC AAABDC ABC	3
i=6	ABC AAA	1	i=13	ABC AAABDC ABCA	4
i=7	ABC AAAB	2	i=14	ABC AAABDC ABCAB	2

# Next数组的构造过程

```
void getNext()
{
    int j = 0, k = -1;
    Next[0] = -1;
    while(j < tlen)
        if(k == -1 || T[j] == T[k])
            Next[++j] = ++k;
        else
            k = Next[k];
}
```

Animation: <https://people.ok.ubc.ca/ylocet/DS/KnuthMorrisPratt.html>

# 字符串匹配

↓ i指针 = 13  
S = ABCAAABDC**ABCAB**...  
T =           **ABC****A**...  
                  ↑ j指针 = 4

$j = \text{Next}[j] = 1$  // “ABCA” 的前缀和后缀的最大公共长度为1

↓ i指针 = 13  
S = ABCAAABDCABC**A**B...  
                  **ABC****A**...  
T =           **A**BCAA...  
                  ↑ j指针 = 1

# 字符串匹配过程

```
int KMP_Index(){
    int i = 0, j = 0;
    getNext(); //预处理出Next数组

    while(i < slen && j < tlen){
        if(j == -1 || S[i] == T[j]){
            i++; j++; //若当前字符能匹配上, i、j指针均向后移动一个位置
        }
        else
            j = Next[j]; //否则j指针跳转到Next[j]指向的位置【敲黑板】
    }
    if(j == tlen)
        return i - tlen; //如果完美地匹配到了T的最后一个字符, 则返回T在S中的位置
    else
        return -1; //否则说明T在S中不存在, 返回-1
}
```

# 例题1 HDU 1686 - Oulipo

给出两个字符串S和T，长度分别为N和M。

求T在S中出现的次数。

( $1 \leq M \leq 10000$ ,  $1 \leq N \leq 1000000$ )

思路：KMP模板题。与“求T在S中首次出现位置”的区别是：每次完整地成功匹配后不立即返回，而是令 $ans++$ ，然后继续沿着S串匹配。

```
int KMP_Count(){
    int ans = 0;
    int i, j = 0;

    if(slen == 1 && tlen == 1){
        if(S[0] == T[0])
            return 1;
        else
            return 0;
    }
    getNext();
    for(i = 0; i < slen; i++){
        while(j > 0 && S[i] != T[j])
            j = Next[j];
        if(S[i] == T[j])
            j++;
        if(j == tlen){
            ans++;
            j = Next[j];
        }
    }
    return ans;
}
```

## 例题2 HDU 1358 - Period

给出字符串S，求S的所有前缀字符串中具有循环节的字符串，输出它们的长度和周期数。

思路：利用Next数组求循环节。

例：对前缀abcdabcdabcd,  $i = 12$ ,  $\text{Next}[i] = 8$ 。

abcdabcdabcd

abcdabcdabcd

Step1: 可能的循环节长度  $p = i - \text{Next}[i] = 4$  (Why? )。即该前缀的前4个字符。

Step2: 进一步判断: `if(i % p == 0 && i / p > 1) printf("%d %d\n", i, i / p);`

即：如果i是p的整数倍，并且不是它本身（循环一次不算循环），那么p就是它的循环节。输出前缀长度i和周期数i/p。

# 扩展KMP

扩展KMP能求出一个串S所有后缀(即 $s[i...len]$ )和模式串T的最长公共前缀。记录在extend数组中。

例如：

S = ABAABCABDC

T = ABCA

ABCA

ABCA

ABCA

extend[0] = 2

extend[1] = 0

extend[2] = 1

extend[3] = 4

# 例题3 HDU 4333 - Revolving Digits

对于一个正整数 $n$ ，将它的最后面若干位平移到最前面，得到一个新的正整数 $m$ 。问在所有可能的 $m$ 中，大于 $n$ 、等于 $n$ 和小于 $n$ 的分别有多少个。

思路：扩展KMP的经典应用。

记所有 $m$ 中大于、等于和小于 $n$ 的个数分别为 $G$ ， $E$ 和 $L$ 。初始化 $G = E = L = 0$ 。

将 $n$ 视为一个字符串。将该字符串复制一遍接在自身后面得到一个新串。利用扩展KMP求出新串每个后缀与原串的最长公共前缀。当公共前缀长度等于原串长度时， $E++$ ；否则只要比较公共前缀的下一位就能确定这个串与原串的大小关系。



# 例题3 HDU 4333 - Revolving Digits

对于一个正整数n，将它的最后面若干位平移到最前面，得到一个新的正整数m。问在所有的可能的m中，大于n、等于n和小于n的分别有多少个。

例：对于n= '343' ，将其复制一遍放在前面得到 '343343' 作为主串S，本身作为模式串T。

i	0	1	2	3	4	5
S[i, ..., n]	<b>343343</b>	43343	<b>3343</b>	<b>343</b>	43	<b>3</b>
T	<b>343</b>	343	<b>343</b>	<b>343</b>	343	<b>343</b>
extend[i]	3	0	1	3	0	1
比较大小	extend[i]==tlen, E++	S[i+extend[i]]>T[extend[i]], G++	S[i+extend[i]]<T[extend[i]], L++	-	-	-

# 例题3： HDU 4333 - Revolving Digits

注意：在计数时相同的串算作一个串，因此要去掉重复串（移动的位数不同但得到的结果相同的串）。

不难发现，只有当这个串有循环节的时候才会产生重复串，因此用KMP的next数组求出最小循环节，用长度除以最小循环节得到循环节个数，再令G、E和L分别除以循环节个数即可。

例如：对于 $n = '123123123'$ ，移动后的所有结果如右图所示。

计算得到的 $L=0$ ， $E=3$ ， $G=6$ 。又由next数组计算得 $n$ 的循环节数量为3，故将这三个数都分别除以3，得到0,1,2即为答案。

231231231

312312312

**123123123**

231231231

312312312

**123123123**

231231231

312312312

**123123123**

# 进一步学习

## 1. KMP + 概率dp

HDU 3689 - Infinite monkey theorem (CaCO3倾力推荐)

<http://www.matrix67.com/blog/archives/366> 【KMP算法与一个经典概率问题】

## 2. AC自动机

HDU 2222 - Keywords Search (模板题)

HDU 2825 - Wireless Password (AC自动机+状压dp)

<https://www.cnblogs.com/cmmdc/p/7337611.html> 【AC自动机详解】

# 参考资料

- [1] <https://www.youtube.com/watch?v=2ogqPWJSftE>
- [2] [https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt\\_algorithm](https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm)
- [3] <https://www.cnblogs.com/yjiyjige/p/3263858.html>
- [4] <http://www.matrix67.com/blog/archives/115>

又称单词查找树，Trie树，是一种**树形结构**。典型应用是用于统计，排序和保存大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。

它的优点是：**利用字符串的公共前缀来减少查询时间**，最大限度地减少无谓的字符串比较，查询效率较高。

<https://segmentfault.com/a/1190000008877595> 一篇好的分析

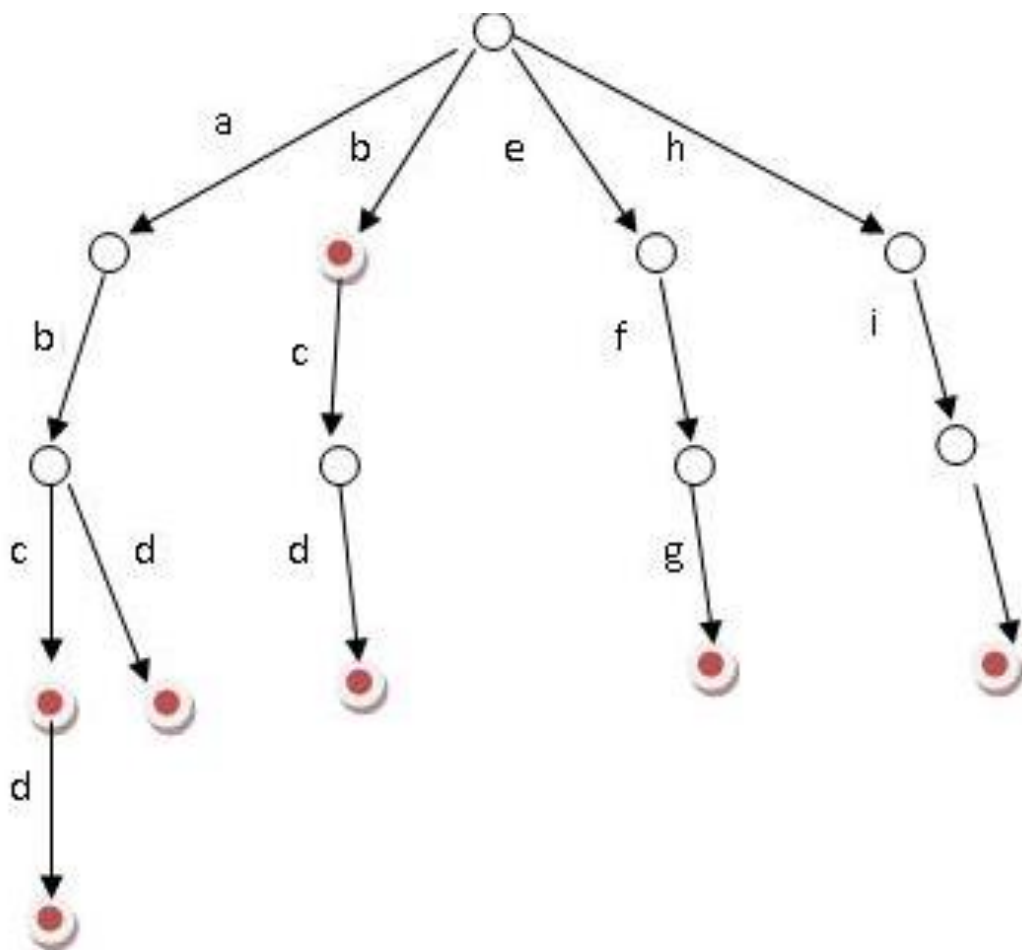
### 基本性質：

根节点不包含字符，除根节点外每一个节点都只包含一个字符；

从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串；

每个节点的所有子节点包含的字符都不相同。

字符串



用边表示字符，每个叶结点表示某一字符串的结尾，可以用bool数组标记。

如果是查询前缀出现的次数的话，那就在开一个sum[]，表示位置i被访问过的次数

如果是普通字符串，每个结点最多26个分支，但形式不局限同样也可以构建01串，带其他字符的串

数组设为tree[maxn][maxm];

maxn: 串的最长位数×字符种类

maxm: 字符种类



字符串

建树

结点编号是随机的，由输入字符串的顺序决定。

tree数组，一维是该结点编号，二维是哪一个孩子

```
int tot=0;
void insert(){ //插入单词s
    len=strlen(s); //单词s的长度
    root=0; //根节点编号为0
    for(int i=0;i<len;i++){
        int id=s[i]-'a'; //字符编号
        if(!trie[root][id]) //如果之前没有从root到id的前缀
            trie[root][id]=++tot; //插入，tot即为第一种编号
        root=trie[root][id]; //顺着字典树往下走
    }
}
```




字符串

查詢

查询某个字符串存不存在，可以是前缀也可以是一整个字符串。

```
bool find(){
    len=strlen(s);
    root=0; //从根结点开始找
    for(int i=0;s[i];i++){
        int x=s[i]-'a';//
        if(trie[root][x]==0)
            return false; //以root为头结点的x字母不存在，返回0
        root=trie[root][x]; //为查询下个字母做准备，往下走
    }
    return true; //找到了
}
```





字符串

## UVALive - 3942

题意：给若干个字符串，问构成目标字符串的方案数

题解：字典树+DP

在插入字符串的时候就标记单词结束位置。

$dp[i]$ 表示以 $i$ 开头的字符串分解的方法数。

状态转移方程： $dp[i] = \sum(dp[i + \text{len}(x)])$ ， $x$ 为 $S[i \dots L]$ 的前缀。



字符串

## UVA - 11488

题意：给定一个字符串集合 $S$ ，定义 $P(S)$ 为所有字符串的公共前缀长度与 $S$ 中字符串个数的乘积，比如 $P(\{000,001,0011\}) = 6$ 。给 $N$ 个01串，从中选出一个集合 $S$ ，使得 $P(S)$ 最大

题解：字典树

建立Trie树，插入的过程中记录经过该节点的字符串个数，更新最大值

# 字符串

## UVA - 11732

题意：给你n个单词N ( $0 < N < 4001$ ，每个最长1000)，两两比较，要求他们运用strcmp时，问进行比较的次数。

题解：做法很多。可以字典树+邻接表分支

```
int strcmp(char *s, char *t)
{
    int i;
    for (i=0; s[i]==t[i]; i++)
        if (s[i]=='\0')
            return 0;
    return s[i] - t[i];
}
```

```
void insert(char *tmp){
    int root=0;
    int len=strlen(tmp);
    for(int i=0;i<=len;i++){
        char id=tmp[i];
        bool f=true;
        int t,j;
        for(j=first[root];j!=-1;j=str[j].nxt){
            if(str[j].ch==id){
                ans+=str[j].val*2;
                f=false;
                t=j;
            }
            else ans+=str[j].val;
        }
        if(f){
            j=cot++;
            str[j].ch=id;
            str[j].val=0;
            str[j].nxt=first[root];
            first[root]=j;
        }else j=t;
        root=j;
        str[root].val++;
    }
}
```




**HDU1298**

**HDU1305**

**POJ 1451**

**POJ 3764**



字符串

## 回文串

**回文串**：即正着读和反着读都是一样的字符串。

常遇见的题目是判断最长回文子串（连续的），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`

传统思路：遍历每一个字符，以该字符为中心向两边查找。其时间复杂度为 $O(n^2)$ 。



# Manacher 馬拉車

预处理：

在字符串首尾，及各字符间各插入一个字符（前提这个字符未出现在串里）  
这样得到的新串长度一定为奇数。

原串

a	a	a	b	a
---	---	---	---	---

新串

#	a	#	a	#	a	#	b	#	a	#
---	---	---	---	---	---	---	---	---	---	---

$Len[i]$ 表示以字符 $T[i]$ 为中心的最长回文子串的最右字符到 $T[i]$ 的长度。



## len 数组

$Len[i]$  表示以字符  $T[i]$  为中心的最长回文子串的最右字符到  $T[i]$  的长度。

性质:  $Len[i]-1$  是该回文子串在原字符串  $S$  中的长度。

新串

#	a	#	a	#	a	#	b	#	a	#
---	---	---	---	---	---	---	---	---	---	---

len

1	2	3	4	3	2	1	4	1	2	1
---	---	---	---	---	---	---	---	---	---	---

所有的回文字串的长度都为奇数，那么对于以  $T[i]$  为中心的最长回文字串，其长度就为  $2*Len[i]-1$ ，经过观察可知， $T$  中所有的回文子串，其中分隔符的数量一定比其他字符的数量多1

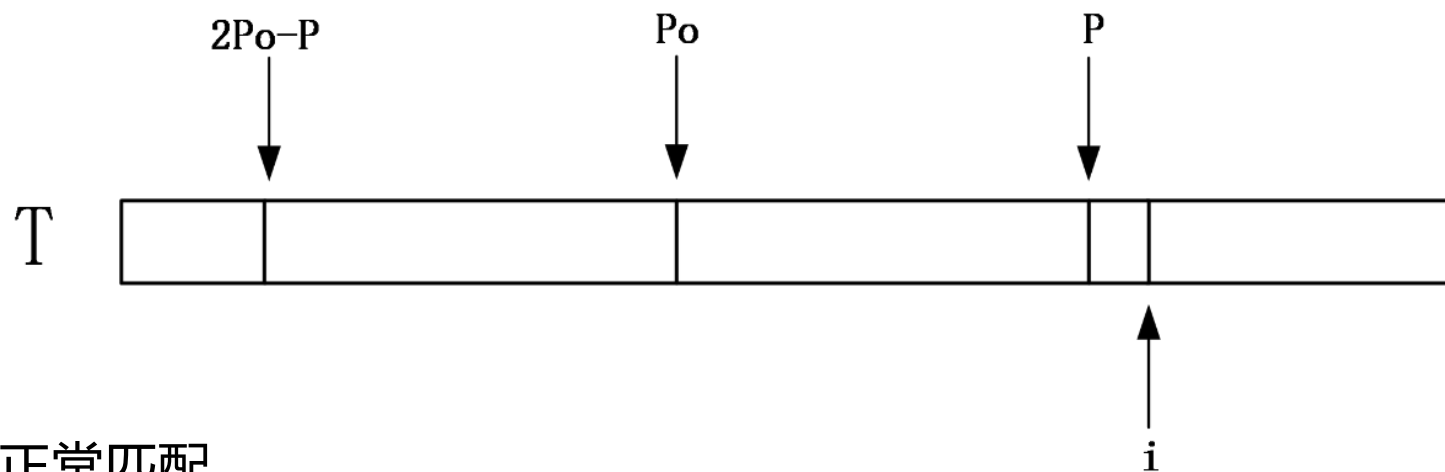
字符串

len数组  
计算

首先从左往右依次计算 $Len[i]$ ，当计算 $Len[i]$ 时， $Len[j](0 \leq j < i)$ 已经计算完毕。

设： $P$ 为之前计算中最长回文子串的右端点的最大值，并且设取得这个最大值的位置为 $po$ ，分两种情况：

第一种情况： $i > P$



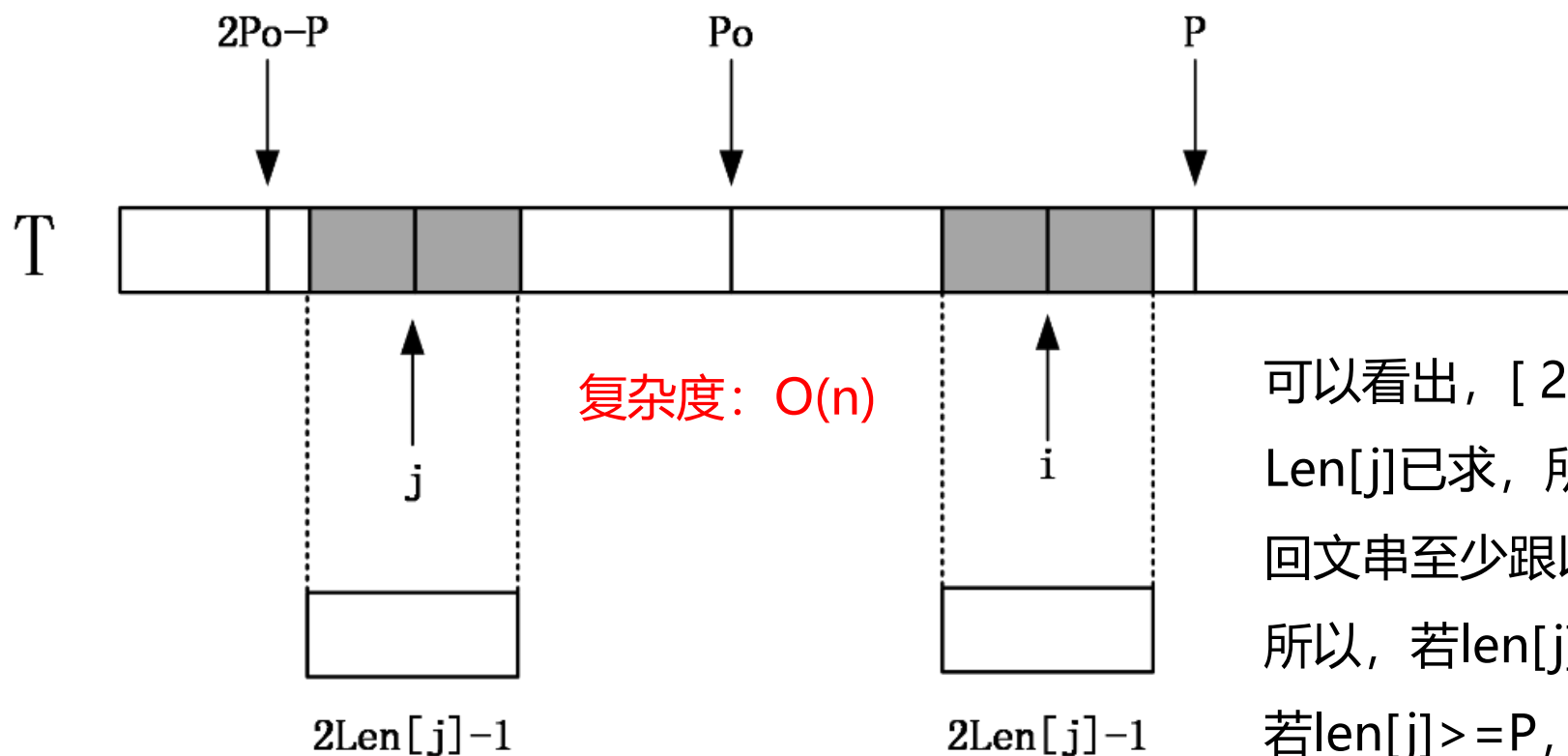
老老实实正常匹配。



字符串

len数组  
计算

第二种情况:  $i \leq P$



可以看出,  $[2P_o - P, P_o)$  和  $(P_o, P]$  是对称的。  
Len[j]已求, 所以在该范围内, 以i为中心的回文串至少跟以j为中心的一样长。  
所以, 若  $\text{len}[j] < P - i$ , 那么  $\text{len}[i] = \text{len}[j]$ ;  
若  $\text{len}[j] \geq P$ , 则越界部分——匹配。

## 字符串

## 代码

```
char str[maxn];
char s_new[maxn<<1];
int p[maxn<<1];

int Init(){
    int len=strlen(str);
    s_new[0]='$';//越界判断
    s_new[1]='#';
    int j=2;
    for (int i=0;i<len;i++){
        s_new[j++]=str[i];
        s_new[j++]='#';
    }
    s_new[j]='\0';
    return j;
}
```

```
int manacher(){
    int len=Init(); // 取得新字符串长度并完成向 s_new 的转换
    int max_len=-1; // 最长回文长度
    int id;//Po: 取得这个最大值的位置
    int mx=0;//P: 之前计算中最长回文子串的右端点的最大值
    for(int i=1;i<len;i++){
        if (i<mx) //第二种情况
            p[i]=min(p[2*id-i],mx-i);
        else //第一种情况
            p[i]=1;
        while(s_new[i-p[i]]==s_new[i+p[i]]) //如果需要, 继续匹配
            p[i]++;
        if (mx<i+p[i]){//更新po,P
            id=i;
            mx=i+p[i];
        }
        max_len=max(max_len,p[i]-1);
        //num[p[i]-1]++;
    }
    return max_len;
}
```

## BZOJ 3790

题意：给定目标字符串，有两种操作，第一种操作可以生成任意回文串，第二种操作可以连接两个串，特殊的，如果第一个串的后缀和第二个串的前缀相同，可以重叠，问最少需要多少次第二种操作。

题解：manacher+贪心

先用马拉车跑出len数组，然后记录每个回文串可以覆盖的区间，以左端点排序，首先选择覆盖的最长回文串，记录位置pos，然后向右遍历，判断该位置回文串的左端点是否大于pos，过程中记得更新最大右位置tmp，不满足时ans++，pos=tmp。



字符串

## HDU - 5677

题意：给n个字符串，求是否能够找出m个回文子串使得它们的长度之和为L

题解：manacher+DP

now和last表示了当前状态和上一个状态

$dp[i][j][i]$ 表示是否能用j个回文子串使得总长度为i

```
while(n--){
    scanf("%s",str);
    manacher(str);
}
int now=0,last=1;
dp[now][0][0]=true;
for(int I=1;I<=100;I++){
    swap(now,last);
    memset(dp[now],0,sizeof(dp[now]));
    for(int i=0;i<=l;i++)
        for(int j=0;j<=k;j++){
            for(int a=0;a<=num[I];a++){
                if(j+a<=k&&i+a*I<=l)
                    dp[now][j+a][i+a*I]=dp[last][j][i];
            }
        }
}
if(dp[now][k][l]) printf("True\n");
else printf("False\n");
```

