# 序列自动机

自动机定义：[https://oi-wiki.org/string/automaton/](https://oi-wiki.org/string/automaton/)

next[i] [j] 表示再原串 S下，第i位后第一个j出现的位置，设串长为n ,字符集大小为a， 预处理时间复杂度为 O（n*a），代码&例题如下

Give a string S and N string $T_i$ , determine whether $T_i$ is a subsequence of S.

If $t_i$ is subsequence of S, print YES,else print NO.

If there is an array $\{K_1,K_2,K_3......K_m\}$ so that and $S_{K_i}=T_i$ , $(1\le i\le m)$, then is a subsequence of S.

Input
The first line is one string S,length(S) ≤100000

The second line is one positive integer N,N≤100000

Then next n lines,every line is a string $T_i$, length( $T_i$) ≤1000

Output
Print N lines. If the i-th $T_i$ is subsequence of S, print YES, else print NO.

样例输入复制

abcdefg
3
abc
adg
cba
样例输出复制

YES
YES
NO

next [maxn] [26]:表示第i个字符后面第一次出现字符j（a-z用0-25表示）的位置。

我们从后往前求，now[j]:字符j从后往前数最晚出现的位置（now数组初始化为-1）

对于每个i，我们都用now[0-25]来更新nex[i][0-25]的值

每经过一个i，我们就更新now数组~使得now数组表示的是最新的状态

但是两个字符串开始的字符是相等的，就没法判断，因为nex[0][j]表示的是0位置后的第一个出现j的位置。所以就要先判断

int loc=now[s[0]-'a']

若loc不为-1，

```
void init()
{
    memset(now,-1,sizeof(now));
    int len=strlen(s);
```

```
    for(int i=len-1; i>=0; i--)
    {
        for(int j=0; j<26; j++)
        {
            nex[i][j]=now[j];
        }
        now[s[i]-'a']=i;
    }
}

int main()
{
    int n,len,loc,flag;
    scanf("%s",s);
    init();
    scanf("%d",&n);
    for(int i=1; i<=n; i++)
    {
        scanf("%s",p);
        len=strlen(p);
        loc=now[p[0]-'a'];
        if(loc==-1)
            printf("NO\n");
        else
        {
            flag=0;
            for(int i=1; i<len; i++)
            {
                loc=nex[loc][p[i]-'a'];
                if(loc==-1)
                {
                    flag=1;
                    break;
                }
            }
            if(!flag)
                printf("YES\n");
            else
                printf("NO\n");
        }
    }
}
```

# 最小表示法

最小表示法用于解决字符串最小表示问题,首先需要明确字符串循环同构，当一个字符串S[1...i]=S[i+1....n]=T时，则说明S与T循环同构

字符串 S的最小表示为与S循环同构的所有字符串中字典序最小的字符串

## 暴力做法

我们每次比较i，j开始的循环同构，把当前比较到的位置记作k，每次遇到不一样的字符时便把大的跳过，最后剩下的就是最优解。

显然这种方法的时间复杂度超级高，在最坏情况下将达到 O( $n^2$)

## 最小表示法

对于字符串S中的一对子串A，B。其在S中的起始位置分别为i,j，且它们的前k个字符均相同，即

$$A[i...i+k-1] == B[j...j+k-1]$$

我们首先考虑A[i+k] > B[j+k]的情况（根据定义要求所有字符串中字典序最小的字符串）

我们发现起始下标为L（i<L<i+k）均不能成为答案。

因为对于任意一个字符串 $S_{i+p}$ （表示以 i+p 为起始位置的字符串）一定存在字符串 $S_{j+p}$ 比它更优。

### 算法流程

1. 初始化指针i=0，j=1，初始匹配长度k=0；
2. 比较第k位的大小，根据比较结果跳转相应指针。若跳转后两个指针相同，则随意选一个加一以保证比较的两个字符串不同
3. 重复上述过程直到比较结束
4. return min（i，j）

### 代码

```
int mininum(char sec[],int n) {
    int k = 0, i = 0, j = 1;
    while (k < n && i < n && j < n) {
        if (sec[(i + k) % n] == sec[(j + k) % n]) {
            k++;//相同继续向后扩展
        } else {//不同根据sec[(i + k) % n] 和 sec[(j + k) % n]的大小对i，j进行跳转变
化
            sec[(i + k) % n] > sec[(j + k) % n] ? i = i + k + 1 : j = j + k + 1;
            if (i == j) i++;//保证起点不同
            k = 0;//k重新归零
        }
    }
    i = min(i, j);
    return i;
}
```

# KMP

视频1 BV1Px411z7Yo

视频2 BV1hW411a7ys

博客 https://blog.csdn.net/v_july_v/article/details/7041827

### 字符串暴力匹配

暴力匹配现有一个文本段str，和一个模式串p，现在要在str中找到模式串p的起始位置，那么如果采用暴力匹配的方法

- 如果当前字符匹配成功（即str[i]==p[i]）,则比较str[++i]与p[++i],继续进行匹配
- 如果当前字符匹配失败（str[i]！=p[i]），令i = i - (j - 1)，j = 0。相当于每次匹配失败时，i 回溯，j 被置为0。

暴力匹配代码如下

```
int check(string str,string p){
    int lens=str.length();
    int lenp=p.length();
    int i=0,j=0;
    while(i<lens){
        if(str[i]==p[i]){
            i++;
            j++;
        }
        else{
            i=i-j-1;
            j=0;
        }
    }
    if(j==lenp)
        return i-j;
    else
        return -1;

}
```

假如按照这个方法进行匹配，假设str={"ABCDABABCDCDCD"} p={"CDC"}，整个过程如下

```
1. S[0]为A，P[0]=C,显然失配，执行2操作
```

2. S[1]与P[0]失继续执行2操作，直到i=2时，str[2]==p[0],此时执行1操作匹配p模式串的第二位，但是在匹配p模式串第三位时str为A，p则为C，失配，则继续
3. 当i=8，j=0时匹配，执行1操作，发现完全匹配，j==lenp，返回当前位置

我们在本次暴力匹配中发雄安，文本串与模式串在i=2与i=8时匹配，i=2时，str[4]!=p[2],文本串回溯至str[3]模式串回溯到p[0]，而str[3]必定与p[0]失配，因为在之前我们已经得知str[3]==p[1]='D'，而p[0]=='C',那有没有一种算法，让i 不往回退，只需要移动j 即可呢?

## 字符串KMP快速匹配

Knuth-Morris-Pratt 字符串查找算法，简称为 "KMP算法"，常用于在一个文本串S内查找一个模式串 P 的出现位置，这个算法由Donald Knuth、Vaughan Pratt、James H. Morris三人于1977年联合发表，故取这3人的姓氏命名此算法。

下面先直接给出KMP的算法流程:

- 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j = -1，或者当前字符匹配成功（即S[i] == P[j]），都令i++，j++，继续匹配下一个字符;
    - 如果j != -1，且当前字符匹配失败（即S[i] != P[j]），则令 i 不变，j = next[j]。此举意味着失配时，模式串P相对于文本串S向右移动了j - next [j] 位。
        - 换言之，当匹配失败时，模式串向右移动的位数为：失配字符所在位置 - 失配字符对应的next 值（next 数组的求解会在下文详细阐述），即**移动的实际位数为：j - next[j]**，且此值大于等于1。

很快，你也会意识到next 数组各值的含义：代表当前字符之前的字符串中，有多大长度的相同前缀后缀。例如如果next [j] = k，代表j 之前的字符串中有最大长度为*k*

此也意味着在某个字符失配时，该字符对应的next 值会告诉你下一步匹配中，模式串应该跳到哪个位置（跳到next [j] 的位置）。如果next [j] 等于0或-1，则跳到模式串的开头字符，若next [j] = k 且 k > 0，代表下次匹配跳到j 之前的某个字符，而不是跳到开头，且具体跳过了k 个字符。

下面简述next数组的计算方法，next数组是从 S[0到i-1]前子串 的前缀后缀最大值

1） 若p[k] == p[j]，则next[j + 1 ] = next [j] + 1 = k + 1；
2） 若p[k] ≠ p[j]，如果此时p[next[k]] == p[j]，则next[ j + 1 ] =  next[k] + 1，否则继续递归前缀索引 k = next[k]，而后重复此过程。

KMP模板：

```
inline void Getnext(string str) { //求出next数组//next数组是从 S[0到i-1]前子串 的前缀后缀最大值
    int t1 = 0, t2, len2 = str.length();
    nxt[0] = t2 = -1;
    while (t1 < len2) {
        if (t2 == -1 || str[t1] == str[t2]) //类似于KMP的匹配
            nxt[++t1] = ++t2;
        else t2 = nxt[t2]; //失配
    }
}
```

```
inline void KMP(string str, string p) { //KMP
    int t1 = 0, t2 = 0, len1 = str.length(), len2 = p.length(); //从0位开始匹配
    while (t1 < len1) { //临界值
        if (t2 == -1 || str[t1] == p[t2]) //匹配成功，继续
            t1++, t2++;
        else t2 = nxt[t2]; //失配
        if (t2 == len2) printf("%d\n",t1-len2+1), t2 = next[t2]; //t2==lenn2时，匹配成功；t1-len2+1即为第一个字母的位置，匹配后t2置为next[t2]
    }
}
```