# Hash

将输入映射到一个值域较小、可以方便比较的范围。

错误率：若进行n次比较，每次错误率为$\frac{1}{m}$,总错误率就为$1-\left(1-\frac{1}{m}\right)^n$

```
typedef unsigned long long ULL;
const int N = 100010, P = 131;
int n, m;
char str[N];
ULL h[N], p[N];
ULL get(int l, int r){
    return h[r] - h[l - 1] * p[r - l + 1];
}
void run(){
     p[0] = 1;
    for(int i = 1; i <= n; i++){
        p[i] = p[i - 1] * P;
        h[i] = h[i - 1] * P + str[i];
    }
}
```

## 字符串匹配

求出模式串的哈希值后，求出文本串每个长度为模式串长度的子串的哈希值，分别与模式串的哈希值比较即可。

## 最长回文子串(nlogn)

二分答案，判断是否可行时枚举回文中心（对称轴），哈希判断两侧是否相等。需要分别预处理正着和倒着的哈希值。

## 允许K次失配的字符串匹配(m+knlog2m)

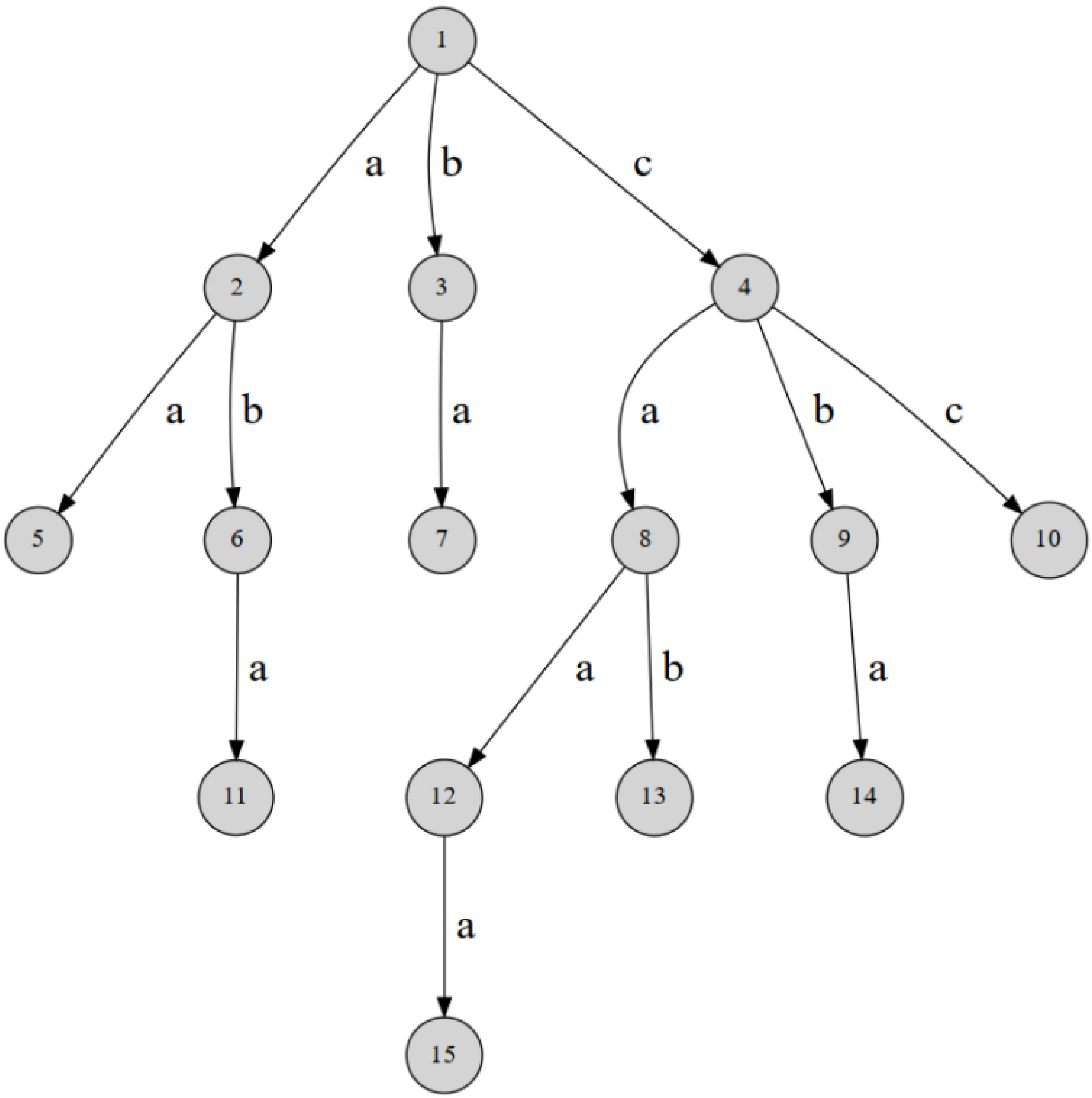
问题：给定长为n的源串，以及长度为m的模式串p，要求查找源串中有多少子串与模式串匹配。s与 s'匹配，当且仅当 s与 s'长度相同，且最多有 k个位置字符不同。其中 1<=n,m<=1e6,0<=k<=5。

## 最长公共子字符串

问题：给定m个总长不超过n的非空字符串，查找所有字符串的最长公共子字符串，如果有多个，任意输出其中一个。其中1<=m,n<=1e6。

## 确定字符串中不同子字符串的数量

# 字典树 (Trie)



Trie

```
const int N=1e5+10;
struct trie{
    int tr[N][26],tot=0;
    bool exist[N];
    void insert(char *s,int len){
        int p=0;
        for(int i=0;i<len;i++){
            if(tr[p][s[i]-'a']==0) tr[p][s[i]-'a']=++tot;
            p=tr[p][s[i]-'a'];
        }
    }
    bool find(char *s,int len){
        int p=0;
        for(int i=0;i<len;i++){

            if(tr[p][s[i]-'a']==0) return 0;
```

```
            p=tr[p][s[i]-'a'];
        }
        return exist[p];
    }
};
```

## 检索字符串

查找一个字符串是否在“字典”中出现过。

## 维护异或极值

将数的二进制表示看做一个字符串，就可以建出字符集为{0,1}的 trie 树

[P4551 最长异或路径 - 洛谷 | 计算机科学教育新生态 (luogu.com.cn)]

```
#include<bits/stdc++.h>
using namespace std;
const int N=1e6+10;
struct edge{
    int to,nxt,val;
}d[N];int head[N],Cnt=1;
void add(int u,int v,int w){d[++Cnt]=(edge){v,head[u],w},head[u]=Cnt;}
int n,val[N];int maxi=0;int ans=0,t[N][2],tot=0;
void insert(int x){
    int p=0;
    for(int i=31;i>=0;i--){
        int now=(x&(1<<i))==0?0:1;
        if(t[p][now]==0) t[p][now]=++tot;
        p=t[p][now];
    }
}
void find(int x){
    int p=0,tp=0;
    for(int i=31;i>=0;i--){
        int now=(x&(1<<i))==0?0:1;
        if(t[p][now^1]) p=t[p][now^1],tp+=1<<i;
        else p=t[p][now];
    }
    maxi=max(maxi,tp);
}
void dfs(int x){
    for(int i=head[x];i;i=d[i].nxt){
        int v=d[i].to,w=d[i].val;
        val[v]=w^val[x];
        insert(val[v]);
        find(val[v]);
        dfs(v);
    }
}
int main(){
    scanf("%d",&n);insert(0);
    for(int i=1;i<n;i++){
```

```
        int v,u,w;
        scanf("%d%d%d",&v,&u,&w);
        add(v,u,w);
    }
    dfs(1);
    cout<<maxi<<endl;
}
```

### 01-trie 合并

- 如果 `a` 没有这个位置上的结点，新合并的结点就是 `b`
- 如果 `b` 没有这个位置上的结点，新合并的结点就是 `a`
- 如果 `a`, `b` 都存在，那就把 `b` 的信息合并到 `a` 上，新合并的结点就是 `a`，然后递归操作处理 a 的左右儿子。

# Manacher算法 O(n)

在原字符串的每个相邻两个字符中间插入一个分隔符，同时在首尾也要添加一个分隔符

| 原串S | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | a | a | a | b | a | | | | | | |

| 转换后得到的串T | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | # | a | # | a | # | a | # | b | # | a | # |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 转换后得到的串T | # | a | # | a | # | a | # | b | # | a | # |
| Len | 1 | 2 | 3 | 4 | 3 | 2 | 1 | 4 | 1 | 2 | 1 |

Len[i]表示以字符T[i]为中心的最长回文字串的最右字符到T[i]的长度；Len[i]表示以字符T[i]为中心的最长回文字串的最右字符到T[i]的长度；Len[i]-1就是该回文子串在原字符串S中的长度。

1.从前向后推，记录右边界最靠右的回文串记为maxright，中心为mid，当前枚举i一定大于mid

2.若i<maxright：找到i关于mid的对称点j=2*mid-maxright,若以j为中心的最大回文串在边界内 len[i]=len[j],若超过，以i为中心的回文串不会超过右边界（证明矛盾）。p[i]大于等于min(p[j],mr-i);

3.若i>=maxright:p[i]从1开始

```
#include<bits/stdc++.h>
using namespace std;
const int N=2e7+10;
char s[N];
char s_new[N];
int p[N];
```

```
int Init()
{
    int len = strlen(s);
    s_new[0] = '$';
    s_new[1] = '#';
    int j = 2;
    for (int i = 0; i < len; i++)
    {
        s_new[j++] = s[i];
        s_new[j++] = '#';
    }

    s_new[j++] = '^';
    return j;
}

int Manacher()
{
    int len = Init();
    int max_len = -1;

    int id;//id为i和j的中心点 i以 id 为对称点翻折到j的位置
    int mx = 0;// mx 代表以 id 为中心的最长回文的右边界

    for (int i = 1; i < len; i++)
    {
        if (i < mx)
            p[i] = min(p[2 * id - i], mx - i); // 2 * id - i 为 i 关于 id 的对称点
        else
            p[i] = 1;

        while (s_new[i - p[i]] == s_new[i + p[i]])
            p[i]++;

        if (mx < i + p[i])
        {
            id = i;
            mx = i + p[i];
        }

        max_len = max(max_len, p[i] - 1);// p[i]-1 即为原字符串中最长回文串的长度
    }

    return max_len;
}

int main()
{

    scanf("%s", s);
    cout<<Manacher()<<endl;


    return 0;
```

```
}
```

# KMP算法

next数组：子串s最长的相等的真前缀与真后缀的长度。

> 举例来说，对于字符串 `abcabcd` ，
>
> $\pi[0]=0$，因为 `a` 没有真前缀和真后缀，根据规定为 0
>
> $\pi[1]=0$，因为 `ab` 无相等的真前缀和真后缀
>
> $\pi[2]=0$，因为 `abc` 无相等的真前缀和真后缀
>
> $\pi[3]=1$，因为 `abca` 只有一对相等的真前缀和真后缀： `a` ，长度为 1
>
> $\pi[4]=2$，因为 `abcab` 相等的真前缀和真后缀只有 `ab` ，长度为 2
>
> $\pi[5]=3$，因为 `abcabc` 相等的真前缀和真后缀只有 `abc` ，长度为 3
>
> $\pi[6]=0$，因为 `abcabcd` 无相等的真前缀和真后缀
>
> 同理可以计算字符串 `aabaaab` 的前缀函数为 $[0,1,0,1,2,2,3]$。

## 前缀函数的朴素算法

枚举1至n-1,计算当前next[i],从最大的真前缀长度开始尝试，如果当前真前缀与真后缀相等，next[i]可求，否则-1直到next[i]=0，i=i+1;

### 优化

**1.相邻的前缀函数值至多增加1 。**

$$\underbrace{\overbrace{s_0\ s_1\ s_2}^{\pi[i]=3}\ s_3}_{\pi[i+1]=4}\ \cdots\ \underbrace{\overbrace{s_{i-2}\ s_{i-1}\ s_i}^{\pi[i]=3}\ s_{i+1}}_{\pi[i+1]=4}$$

**2.最优情况：s[i+1]=s[next[i]] 此时 next[i+1]=next[i]+1;**

**若s[i+1]！=s[next[i]]，需要找到第二大的长度使得 s[0]...s[k] (前缀)==s[kk]...s[i] （s[0,i]的后缀）**

**分析一下 s[0]...s[k-1]==s[kk]....s[i]&&s[k+1]（前缀的下一个字符）==s[i+1]**

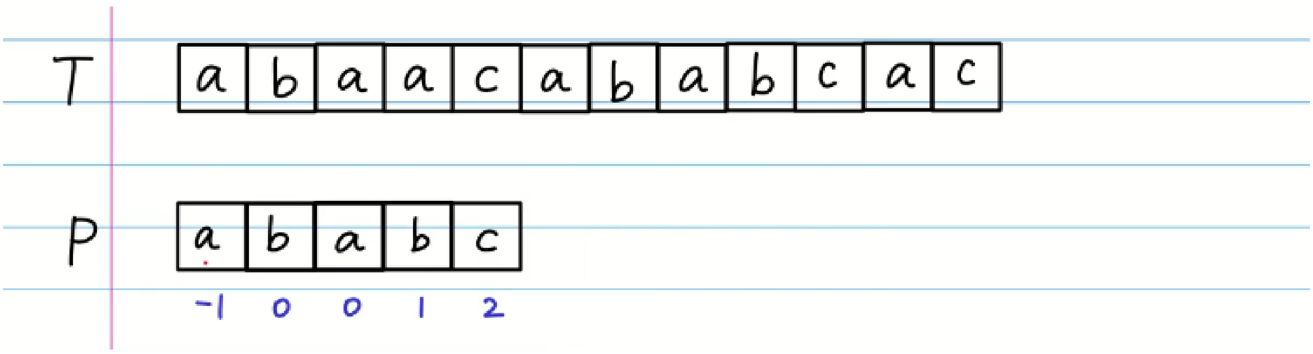
**s[0]..s[k-1]当且仅当第二大只有 k=next[next[i]-1]，如此反复直到k=0;**

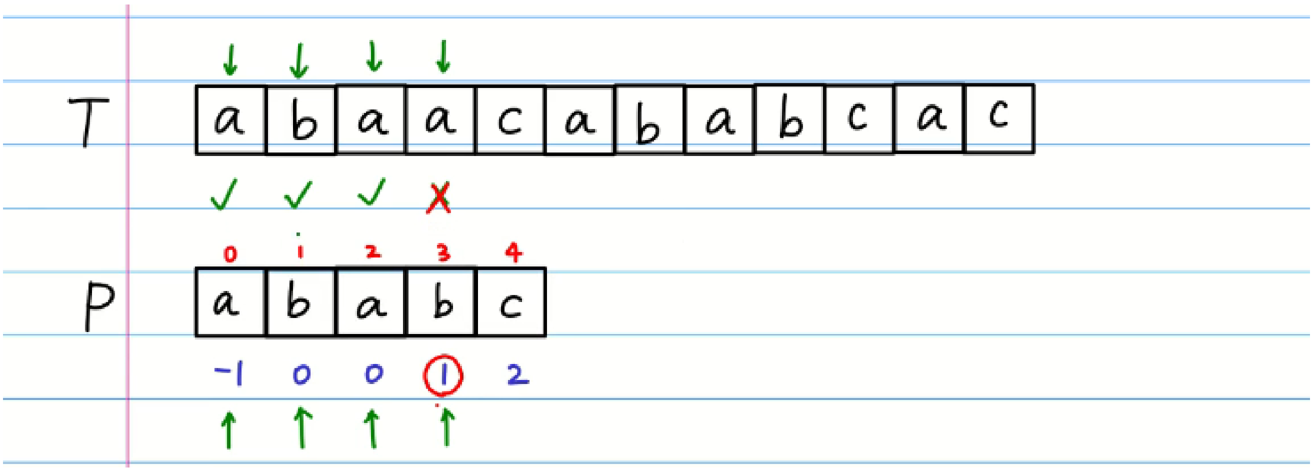
**得出算法**

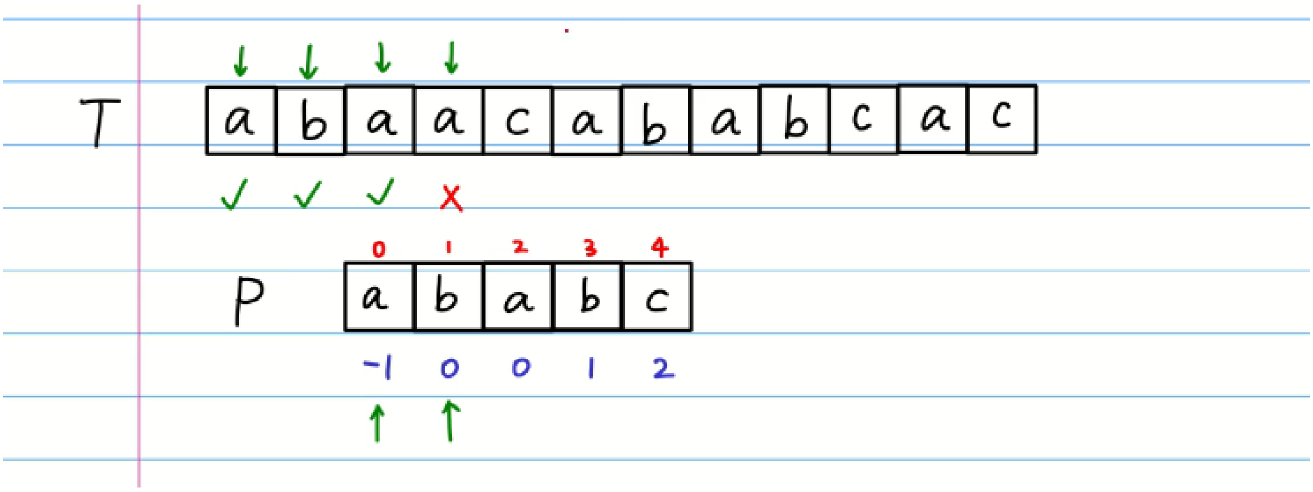
```
vector<int> getnxt(string s) {
  int n = (int)s.length();
  vector<int> pi(n);
  for (int i = 1; i < n; i++) {
    int j = pi[i - 1];
    while (j > 0 && s[i] != s[j]) j = pi[j - 1];
    if (s[i] == s[j]) j++;
    pi[i] = j;
  }
  return pi;
}
void KMP_Count(string s,string p) {//x为模式串
     int m=s.size(),n=p.size();
    for(int i = 0, j = 0; i < m; ++i){
        while(j && s[i] != p[j]) j = nxt[j-1];
        if(s[i] == p[j]) j++;
        if(j == n){
            cout<<i-n+2<<endl;
            j = nxt[j-1];
        }
    }
}
```

## 在字符串中查找子串

**给定一个文本t和一个字符串p，我们尝试找到并展示p在t中的所有出现（occurrence）。**






## 字符串的周期

**n-next[n-1]为字符串s的最小周期**

# 扩展 KMP

对于个长度为n的字符串 。定义函数z[i]表示s和 s[i,n-1]（即以 s[i]开头的后缀）的最长公共前缀（LCP）的长度。 z被称为s的 **Z 函数**。特别地,z[0]=0。

- $z(\mathrm{aaaaa}) = [0, 4, 3, 2, 1]$
- $z(\mathrm{aaabaab}) = [0, 2, 1, 0, 2, 1, 0]$
- $z(\mathrm{abacaba}) = [0, 0, 1, 0, 3, 0, 1]$

下面我们以O（n）的方法求出z函数，算法的过程中我们维护右端点最靠右的匹配段，记[l,r]，初始化l=r=0;

在计算z[i]的过程中,

1.若i<=r,必有s[i,r]==s[i-l,r-l],于是z[i]>=min(z[i-l],r-i+1)。

1.1. z[i-l]<r-i+1,则z[i]=z[i-l]

1.2.z[i-l]>=r-i+1 ,令z[i]=r-i+1，暴力枚举

2.i>r，暴力枚举

3.求出z[i],更新[l,r]。

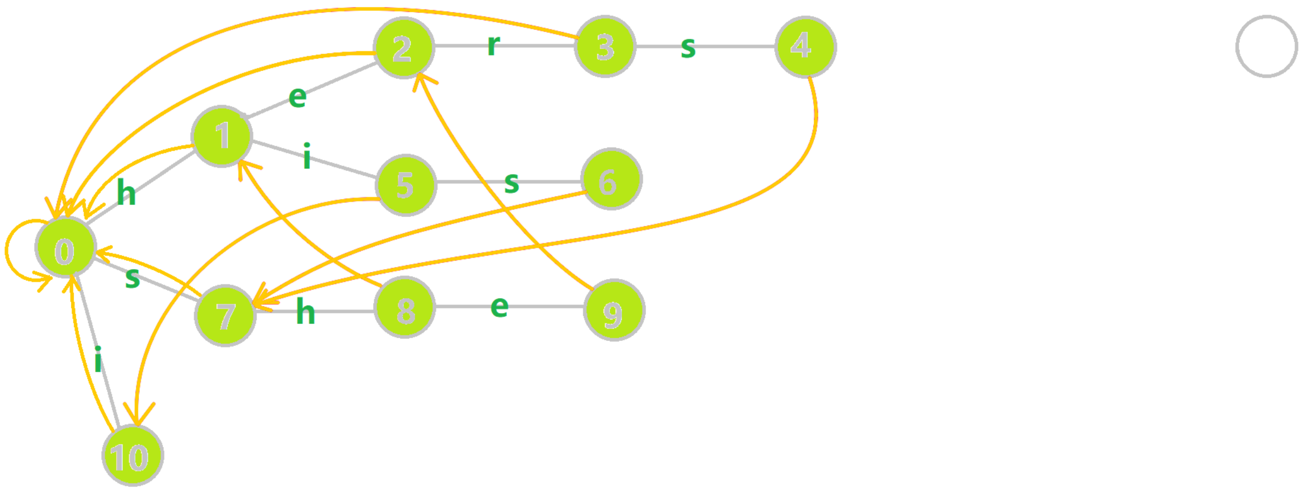
[Z Algorithm (JavaScript Demo) (utdallas.edu)]

# AC自动机

AC自动机的构造:

1. 基础的 Trie 结构：将所有的模式串构成一棵 Trie。
2. KMP 的思想：对 Trie 树上所有的结点构造失配指针。

失配指针：fail 指针指向所有模式串的前缀中匹配当前状态的最长后缀如果当前点匹配失败,则将指针转移到fail指针指向的地方,这样就不用回溯,而可以路匹配下去了.(当前模式串后缀和fail指针指向的模式串部分前缀相同,如abce和bcd,我们找到`c`发现下一个要找的不是`e`,就跳到`bcd`中的`c`处,看看此处的下一个字符(`d`)是不是应该找的那一个)




开始构建fail指针，将tr[u,c]理解为从状态u（结点） 后加一个字符`c`到达的状态（结点）

若当前tr[p,c]=u，且假设深度小于u的所有节点的fail指针都已经得到。

1.若tr[fail[p],c]存在，则让 u 的 fail 指针指向 tr[fail[u],c]。

2.若tr[fail[p],c]不存在，则继续寻找tr[fail[fail[p]],c]。重复1的判断过程知道跳到根节点。

3.若最后没有，指向根节点。

将结点按 BFS 顺序入队，依次求 fail 指针,第一次bfs默认根节点的fail指针指向本身，取出队列首项，遍历26个字母，若tr[u,i]存在，fail[tr[u,i]]=tr[fail[u,i]];否则tr[u,i]指向tr[fail[u,i]]的状态

如果在位置u失配，我们会跳转到fail[u]的位置。所以我们可能沿着 fail 数组跳转多次才能来到下一个能匹配的位置。所以我们可以用tr数组直接记录记录下一个能匹配的位置，这样就能节省下很多时间。

```
namespace AC {
    int tr[N][26], tot;
    int e[N], fail[N];
    void insert(char *s) {
        int u = 0;
        for (int i = 1; s[i]; i++) {
            if (!tr[u][s[i] - 'a']) tr[u][s[i] - 'a'] = ++tot;
            u = tr[u][s[i] - 'a'];
        }
        e[u]++;
    }
    queue<int> q;
    void build() {
        for (int i = 0; i < 26; i++)
            if (tr[0][i]) q.push(tr[0][i]);
        while (q.size()) {
            int u = q.front();
            q.pop();
            for (int i = 0; i < 26; i++) {
                if (tr[u][i])
                    fail[tr[u][i]] = tr[fail[u]][i], q.push(tr[u][i]);
                else
                    tr[u][i] = tr[fail[u]][i];
            }
        }
    }
    int query(char *t) {
        int u = 0, res = 0;
        for (int i = 1; t[i]; i++) {
            u = tr[u][t[i] - 'a'];  // 转移
            for (int j = u; j && e[j] != -1; j = fail[j]) {
                res += e[j], e[j] = -1;
            }
        }
        return res;
    }
    void init(){
        memset(e,0,sizeof e);
        memset(fail,0,sizeof fail);
        memset(tr,0,sizeof tr);
        tot=0;
    }
}  // namespace AC
```