云某人的string

云某人的string

- 基础概念
  - border
  - 周期和循环节
  - border 性质
  - border 树
  - 广义border
  - 失配指针
- 字符串Hash
  - 概念
  - 多项式取模哈希
    - 多项式 Hash
    - 多项式取模 Hash（模哈）
    - Hash 模数
      - 快速求子串 hash
    - 回文串hash性质
    - code
- KMP
  - 模板
  - 题目
    - 串合并去重
    - 反复删除子串
    - 求最长真周期
    - 求不重叠border数目
- trie
  - 模板
  - 题目
    - 字符串子串插入
- ac自动机
  - 模板
  - 题目
    - 子串删除问题
    - 模式串出现次数
    - acam dp
      - P3041 [USACO12JAN] Video Game G
      - P4052 [JSOI2007] 文本生成器
    - 模式串互相匹配

基础概念

border

如果字符串 S 的同长度的前后缀完全相同，则称此前后缀为一个 $Border$，也可以指这个 $Border$ 的长度

周期和循环节

如果一个字符串 Z 在 S 中循环出现

- 如果可能不完整出现，则为循环周期
- 如果完整出现，则为循环节

border 性质

周期定理： 若 p, q 为 S 的周期，则 $\gcd(p, q)$ 也为 S 的周期。
等差数列： 一个串的 border 数量是 (n) 个，但他们组成了 $O(\log n)$ 个等差数列

border 树

i 的父亲为 nxt_i 构成的一棵树（包括 0，0 为根节点）

性质：

- 每个前缀 pre_i 的所有 border 为节点 i 到根的链
- x 的子树有长度为 x 的 border
- 求两个前缀的公共 border 等价于求 lca

广义border

对于字典 D 和串 S，相等长度的 S 后缀和任意一个字典串 T 的前缀称为一个border

失配指针

类似 KMP的border，任意节点的 border长度减一，一点是父节点的border。银川可以通过遍历父节点的失配指针链求解。

因此在求失配指针的时候，一定要按长度从小到大求，即**bfs**

复杂度线性，用类似势能分析来求。

字符串Hash

概念

哈希：一种单射函数，可以将**万物**映射为一个整数值。
字符串哈希： 将一个字符串映射为一个整数值的方法，通常用来比较两个字符串是否相等

性质：

- **必要性：** 若字符串 $S = T$，则 $H(S) = H(T)$
- **非**充分性：若 $H(S) = H(T)$，**不一定** $S = T$ (可以实现，但不要求)

Hash检测： 通过检测 $H(S)$ 和 $H(T)$ 是否相等，来判断 S 和 T 是否相等的方法。
Hash冲突： $H(S) = H(T)$ 时，但 $S \neq T$，则称发生了Hash冲突

> Hash 检测时 Hash 冲突的概率是衡量 Hash 算法好坏的重要指标
>
> > 比如冲突概率 $\leq 10^{-4}$

多项式取模哈希

多项式 Hash

将字符串看作某个进制 (Base) 下的数字串(类似直接 `stoi()`，只是进制不同)

$$H(S) = \sum s_i \times Base^{1+n-i} = \mathbf{H(S[1, |S| - 1]) \times Base + S[|H|]}$$
$$= S_1 \times Bash^{n-1} + S_2 \times Base^{n-2} + \cdots + S_n \times Base^0$$

优点： 字符串和 Hash 值一一对应，**不会**产生Hash冲突，且利用率高
缺点： 数字范围过大，难以用原始数据存储和比较

多项式取模 Hash（模哈）

为了解决多项式 Hash 值域过大的问题，在效率和冲突率中的折中。

将 Hash 值对一个较大的质数取模

$$H'(S) = H(S)\%mod$$

优点： 使得Hash值可以用 `uint` / `ulong` 存储和比较
缺点： 小概率 Hash 冲突

因为当检验次数 $\geq \sqrt{mod}$，有较大概率发生错误。
为了保证冲突率低，模哈使用的 mod 最好超过 Hash 检测次数的平方

Hash 模数

优秀的 Hash 模数首先应满足：**足够大**

自然溢出： 使用 `ULL` 保存Hash值，溢出 `ULL`，等同于 $\%2^{64}$(很容易卡)

优秀的模数还一个是一个 **质数**

质数单模： 选取**一个**$10^9 \sim 10^{10}$ 的大质数，但是有 **广为人知** 的方法构造冲突。

> 如果在int内更多，会更有优势 (在32位计算机上)

双模： 进行多次不同质数的单模哈希，冲突概率为各次单模的概率乘积

> 在不泄露模数的情况下，没有已知方法可以构造冲突

快速求子串 hash

$$H(S) = (S_l \times Bash^{r-l} + S_{l+1} + 1 \times Base^{r-l-1} + \cdots + S_r)\%mod$$

令 $F(i) = H(prefix[i])$

$$F(l-1) = (S_1 \times Bash^{l-2} + S_2 \times Base^{l-3} + \cdots + S_{l-1})\%mod$$
$$F(r) = (S_1 \times Bash^{r-1} + S_2 \times Base^{r-2} + \cdots + S_r)\%mod$$

$$\therefore H(S[l, r]) = F(r) - F(l-1) \times Base^{r-l+1}$$

回文串hash性质

对于回文串，正向Hash和反向Hash值是一样的

code

```
const int base = 26;
PII mod = {1000000321,1000000711};// 模数，随便塞两个素数也行
string st;// 哈希字符串
vector<PII> s(N);// 前缀和

void init(string &st,vector<PII> &s)
{
    int L = st.size();
    for(int i=1;i<=L;++i)
    {
        s[i].X = (s[i-1].X * base) % mod.X;
        s[i].X = (s[i].X + st[i]-'a')%mod.X;
        s[i].Y = (s[i-1].Y * base) % mod.Y;
        s[i].Y = (s[i].Y + st[i]-'a')%mod.X;
    }
}

PII query(int l,int r)
{
    PII res = {0,0};
    res.X = (s[r].X - s[l-1].X*qpow(base,r-l+1,mod.X));
    res.Y = (s[r].Y - s[l-1].Y*qpow(base,r-l+1,mod.Y));
    return res;
}
```

KMP

模板

求 border + kmp查找

```
// 从 2 开始
string st;// 模式串
string s;// 查找串

int nxt[N];
void get_next(string &st)// 求border
{
    for(int i=2;i<=st.size();++i)
    {
        nxt[i] = nxt[i-1];
        while(nxt[i] && st[nxt[i]+1] != st[i])  nxt[i] = nxt[nxt[i]];
        nxt[i] += st[nxt[i]+1] == st[i];
    }
}

int kmp(int l,int r,string& s)// 求第一个匹配的位置
{
    for(int i=l,j=0;i<=r;++i)
    {
        while(j>0 && st[j+1] != s[i])   j = nxt[j];
        j += (st[j+1] == s[i]);
```

```
        if(j == m)  return i-j+1;
    }
}

int kmp(int l,int r,string& s)// 求匹配的个数
{
    int cnt = 0;
    for(int i=l,j=0;i<=r;++i)
    {
        while(j>0 && st[j+1] != s[i])   j = nxt[j];
        j += (st[j+1] == s[i]);
        if(j == m)
        {
            cnt ++;
            j = nxt[j];
        }
    }
}
```

题目

串合并去重

需要合并 a 和 b，并去除两者间公共部分。

只需要对 b 新建border数组，然后用 a 最后长度等于 b 的部分进行匹配，匹配的最大长度及公共部分

对于 n 个串首尾相接

```
int n;
int nxt[N];
string s0,st;

void func(void)
{
    cin >> n;
    cin >> s0;
    for(int i=2;i<=n;++i)
    {
        cin >> st;
        st = '_' + st;
        for(int j=2;j<st.size();++j)
        {
            nxt[j] = nxt[j-1];
            while(nxt[j] && st[j] != st[nxt[j]+1])  nxt[j] = nxt[nxt[j]];
            nxt[j] += (st[j] == st[nxt[j]+1]);
        }
        int p = 0;
        int be = (s0.size() >= st.size()-1 ? s0.size()-st.size()+1 : 0);
        for(int j=be;j<s0.size();++j)
        {
            while(p>0 && st[p+1] != s0[j])  p = nxt[p];
            if(st[p+1] == s0[j])    p ++;
        }
        for(int j=p+1;j<st.size();++j)  s0 += st[j];
    }
```

```
    cout << s0 << '\n';
}
```

反复删除子串

每次删除第一次出现的要求子串

存储各个位置匹配的最大长度，匹配满后跳跃到对应位置可直接从border中匹配

> `string.erase()` 貌似很慢，但是好像都不会tle，为防万一可以用栈存储。

```
void func(void)
{
    string s,t;
    cin >> s >> t;
    int ls = s.size(), lt = t.size();
    t = '_' + t;
    vector<int> nxt(lt+1),p(ls+1);
    for(int i=2;i<=lt;++i)
    {
        nxt[i] = nxt[i-1];
        while(nxt[i] && t[i] != t[nxt[i]+1])     nxt[i] = nxt[nxt[i]];
        nxt[i] += (t[i] == t[nxt[i]+1]);
    }
    for(int i=0,j=0;i<s.size();++i)
    {
        while(j>0 && t[j+1] != s[i])    j = nxt[j];
        if(t[j+1] == s[i])  j ++;
        p[i] = j;
        if(j == lt)
        {
            j = (i >= lt ? p[i-lt] : 0);
            i -= lt;
            s.erase(i+1,lt);
        }
        if(i >= 0)  p[i] = j;
    }
    cout << s << '\n';
}
```

求最长真周期

根据性质：周期 $= |S| - border$
最长**真**周期则用最短 broder，用dfs传递最短border即可。

```
> 这道题求最长真周期和
int n,nxt[N];
string st;
vector<int> v[N];
int ans;

void dfs(int p,int t)
{
    if(!t)  t = nxt[p];
    ans += (t ? p-t : 0);
```

```
    for(auto &i : v[p])
    {
        dfs(i,t);
    }
}

void func(void)
{
    cin >> n >> st;
    st = '_' + st;
    v[0].push_back(1);
    for(int i=2;i<=n;++i)
    {
        nxt[i] = nxt[i-1];
        while(nxt[i] && st[i] != st[nxt[i]+1])  nxt[i] = nxt[nxt[i]];
        nxt[i] += (st[i] == st[nxt[i]+1]);
        v[nxt[i]].push_back(i);
    }
    dfs(0,nxt[0]);
    cout << ans << '\n';
}
```

求不重叠border数目

求各个前缀长度 $\leq S/2$ 的border数目

用dp存储各个串的border数目，每次将前方 $T \leq S/2$ 的dp加入答案

```
int n,ans;
string st;
int nxt[N],dp[N];

void func(void)
{
    cin >> st;
    n = st.size(),ans = 1;
    st = '_' + st;
    memset(dp,0,sizeof dp);
    for(int i=2;i<=n;++i)
    {
        nxt[i] = nxt[i-1];
        while(nxt[i] && st[i] != st[nxt[i]+1])  nxt[i] = nxt[nxt[i]];
        nxt[i] += (st[i] == st[nxt[i]+1]);
        if(nxt[i])  dp[i] = dp[nxt[i]] + 1;
        // cout << i << ' ' << dp[i] << '\n';
    }
    for(int i=n;i>=1;--i)
    {
        int p = nxt[i];
        vector<int> tmp;
        while(p*2 > i)
        {
            tmp.push_back(p);
            p = nxt[p];
        }
```

```
        for(auto &i : tmp)  nxt[i] = p;
        if(p)   ans = ans*(dp[p]+2)%M;
    }
    cout << ans << '\n';
}
```

trie

模板

```
int idx,nxt[N],ext[N];

void insert(string &st,int id)
{
    int p = 0;
    for(int i=0;i<st.size();++i)
    {
        int c = st[i]-'a';
        if(!nxt[p][c])  nxt[p][c] = ++ idx;
        p = nxt[p][c];
    }
    ext[p] = id;
}

bool find(string &st)
{
    int p = 0;
    for(int i=0;i<st.size();++i)
    {
        int c = st[i]-'a';
        if(!nxt[p][c])  return false;
        p = nxt[p][c];
    }
    return ext[p];
}
```

题目

字符串子串插入

需要插入反复插入一个字符串的子串或者部分，每次不能新建整个串，而是从上次插入地方继续加入节点

```
for(auto &i : s0)
{
    if(i == 'B')
    {
        if(st.size())   st.erase(st.end()-1);
        p = fa[p];
    }
    else if(i == 'P')
    {
        for(auto &j : st)
        {
```

```
            int c = j-'a';
            if(!nxt[p][c])  nxt[p][c] = ++ idx;
            trie[p][c] = nxt[p][c];
            fa[nxt[p][c]] = p;
            p = nxt[p][c];
        }
        mp[++ cnt] = p;
        st.clear();
    }
    else    st += i;
}
```

ac自动机

模板

trie树+fail（失配）指针
每个串保存trie树中最长相同前缀，这样失配的时候可以直接跳转匹配。

```
int n,idx;
int nxt[N][26],fail[N];
bitset<N> ext;
// trie
void insert(string &st,int id)
{
    int p = 0;
    for(int i=0;i<st.size();++i)
    {
        int c = st[i] - 'a';
        if(!nxt[p][c])  nxt[p][c] = ++ idx;
        p = nxt[p][c];
    }
    ext[p] = id;
}
// acam
void build_acam(void)
{
    queue<int> q;
    for(int i=0;i<D;++i)
        if(nxt[0][i])   q.push(nxt[0][i]);
    while(q.size())
    {
        int p = q.front();  q.pop();
        for(int i=0;i<26;++i)
        {
            if(!nxt[p][i]) nxt[p][i] = nxt[fail[p]][i];
            else
            {
                int ps = nxt[p][i]; q.push(ps);
                fail[ps] = nxt[fail[p]][i];
            }

        }
    }
```

```
}

void query_acam(string &st)
{
    for(int i=0,p=0;i<st.size();++i)
    {
        int c = st[i]-'a';
        // 因为trie树的节点被重构了，所以可以直接跑
        p = nxt[p][c];
        // do...
    }
}
```

题目

子串删除问题

给定一个字符串 S 和一个字典，反复删除 S 在字典出现的第一个模式串

和前面那道kmp一样，只不过这题需要删除所有模式串的结果。

解法和那题差不多，只不过需要存储匹配的自动机节点，而非border串长度（其实本质是相同的）
这里没有用erase，而是用了栈，刚好两道题可以互相对应 反复删除子串

```
string s0;
int n,idx,tmp[N];
int nxt[N][26],fail[N];
int ext[N];

void insert(string &st,int L)
{
    int p = 0;
    for(int i=0;i<st.size();++i)
    {
        int c = st[i]-'a';
        if(!nxt[p][c])  nxt[p][c] = ++ idx;
        p = nxt[p][c];
    }
    ext[p] = L;
}

void build_acam(void)
{
    queue<int> q;
    q.push(0);
    while(q.size())
    {
        int p = q.front();  q.pop();
        for(int i=0;i<26;++i)
        {
            if(!nxt[p][i])  nxt[p][i] = nxt[fail[p]][i];
            else
            {
                int ps = nxt[p][i];
                fail[ps] = fail[p];
```

```
                while(fail[ps] && !nxt[fail[ps]][i])    fail[ps] =
fail[fail[ps]];
                if(p && nxt[fail[ps]][i])   fail[ps] = nxt[fail[ps]][i];
                q.push(ps);
            }

        }
    }
}

void func(void)
{
    cin >> s0 >> n;
    string st;
    for(int i=1;i<=n;++i)
    {
        cin >> st;
        insert(st,st.size());
    }
    vector<pair<char,int>> ans;
    build_acam();
    int p = 0;
    for(int i=0;i<s0.size();++i)
    {
        int c = s0[i]-'a';
        p = nxt[p][c];

        ans.push_back({s0[i],0});
        if(ext[p])
        {
            for(int j=ext[p];j>0;--j)   ans.pop_back();
            p = (ans.size() ? (ans.back()).Y : 0);
        }
        if(ans.size())  ans.back().Y = p;
    }
    for(auto &i : ans)  cout << i.X;
}
```

模式串出现次数

统计模式串在字典中的出现次数

求fail树上该节点的子树大小，其实也和kmp求出现次数一样。

```
int n,idx;
int nxt[N][26],fail[N];
int cnt[N],mp[N];
vector<int> ft[N];

void insert(string &st,int id)
{
    int p = 0;
    for(int i=0;i<st.size();++i)
    {
        int c = st[i]-'a';
```

```
        if(!nxt[p][c])  nxt[p][c] = ++ idx;
        p = nxt[p][c];
        cnt[p] ++;
    }
    mp[id] = p;
}

void build_acam(void)
{
    queue<int> q;   q.push(0);
    while(q.size())
    {
        int lp = q.front(); q.pop();
        for(int i=0;i<26;++i)
        {
            if(!nxt[lp][i]) nxt[lp][i] = nxt[fail[lp]][i];
            else
            {
                int p = nxt[lp][i];
                fail[p] = fail[lp];
                while(fail[p] && !nxt[fail[p]][i])  fail[p] = fail[fail[p]];
                if(lp && nxt[fail[p]][i])   fail[p] = nxt[fail[p]][i];
                ft[fail[p]].push_back(p);
                q.push(p);
            }
        }
    }
}

void dfs(int p)
{
    for(auto &i : ft[p])
    {
        dfs(i);
        cnt[p] += cnt[i];
    }
}

void func(void)
{
    cin >> n;
    string st;
    for(int i=1;i<=n;++i)
    {
        cin >> st;
        insert(st,i);
    }
    build_acam();
    dfs(0);
    for(int i=1;i<=n;++i)   cout << cnt[mp[i]] << '\n';
}
```

acam dp

P3041 [USACO12JAN] Video Game G

Bessie 在玩一款游戏，该游戏只有三个技能键 `A`，`B`，`C` 可用，但这些键可用形成 n 种特定的组合技。第 i 个组合技用一个字符串 s_i 表示。

Bessie 会输入一个长度为 k 的字符串 t，而一个组合技每在 t 中出现一次，Bessie 就会获得一分。s_i 在 t 中出现一次指的是 s_i 是 t 从某个位置起的连续子串。如果 s_i 从 t 的多个位置起都是连续子串，那么算作 s_i 出现了多次。

若 Bessie 输入了恰好 k 个字符，则她最多能获得多少分?

solution

`dp[i][j]` 表示长度为 i 的以 acam节点j 为结尾字符串最大 dp 值

那么dp每次就可以从acam树上的父节点转移到子节点

```
int n,m,idx;
int nxt[N][3],fail[N],val[N];
int dp[M][M];

void insert(string &st)
{
    int p = 0;
    for(int i=0;i<st.size();++i)
    {
        int c = st[i]-'A';
        if(!nxt[p][c])  nxt[p][c] = ++ idx;
        p = nxt[p][c];
    }
    val[p] ++;
}

void build_acam(void)
{
    queue<int> q;
    for(int i=0;i<3;++i)
    {
        if(nxt[0][i])   q.push(nxt[0][i]);
    }
    while(q.size())
    {
        int lp = q.front(); q.pop();
        for(int i=0;i<3;++i)
        {
            if(!nxt[lp][i]) nxt[lp][i] = nxt[fail[lp]][i];
            else
            {
                int p = nxt[lp][i];
                fail[p] = nxt[fail[lp]][i];
                q.push(p);
                val[p] += val[fail[p]];
            }
        }
    }
```

```
}

void func(void)
{
    cin >> n >> m;
    for(int i=0;i<n;++i)
    {
        string st;  cin >> st;
        insert(st);
    }
    build_acam();
    int ans = 0;
    memset(dp,0xaf,sizeof dp); // -inf
    dp[0][0] = 0;
    for(int i=1;i<=m;++i)
    {
        for(int j=0;j<=idx;++j)
        {
            for(int k=0;k<3;++k)    dp[i][nxt[j][k]] = max(dp[i][nxt[j][k]],dp[i-
1][j]+val[nxt[j][k]]);
        }
    }
    for(int i=0;i<=idx;++i) ans = max(ans,dp[m][i]);
    cout << ans << '\n';
}
```

P4052 [JSOI2007] 文本生成器

JSOI 交给队员 ZYX 一个任务，编制一个称之为“文本生成器”的电脑软件：该软件的使用者是一些低幼人群，他们现在使用的是 GW 文本生成器 v6 版。

该软件可以随机生成一些文章——总是生成一篇长度固定且完全随机的文章。 也就是说，生成的文章中每个字符都是完全随机的。如果一篇文章中至少包含使用者们了解的一个单词，那么我们说这篇文章是可读的（我们称文章 s 包含单词 t，当且仅当单词 t 是文章 s 的子串）。但是，即使按照这样的标准，使用者现在使用的 GW 文本生成器 v6 版所生成的文章也是几乎完全不可读的。ZYX 需要指出 GW 文本生成器 v6 生成的所有文本中，可读文本的数量，以便能够成功获得 v7 更新版。你能帮助他吗?

和上一题不同，求的是子串存在模式串的串总数
solution
求存在不方便，所以求不存在
`dp[i][j]` 表示长度为 i 的以 acam节点j 为结尾的字符串不存在字典串总数

最后再 $sum-dp$ 即可

```
int n,m,idx;
int nxt[N][D],fail[N];
int dp[110][N];
bitset<N> ext;

void insert(string &st)
{
    int p = 0;
    for(int i=0;i<st.size();++i)
    {
        int c = st[i]-'A';
```

```
        if(!nxt[p][c])  nxt[p][c] = ++ idx;
        p = nxt[p][c];
    }
    ext[p] = true;
}

void build_acam(void)
{
    queue<int> q;
    for(int i=0;i<26;++i)
        if(nxt[0][i])   q.push(nxt[0][i]);
    while(q.size())
    {
        int p = q.front();  q.pop();
        for(int i=0;i<D;++i)
        {
            if(!nxt[p][i])  nxt[p][i] = nxt[fail[p]][i];
            else
            {
                int ps = nxt[p][i]; q.push(ps);
                fail[ps] = nxt[fail[p]][i];
                ext[ps] = ext[fail[ps]];
            }
        }
    }
}

void func(void)
{
    cin >> n >> m;
    for(int i=0;i<n;++i)
    {
        string st;  cin >> st;
        insert(st);
    }
    build_acam();
    dp[0][0] = 1;
    for(int i=1;i<=m;++i)
    {
        for(int j=0;j<=idx;++j)
        {
            for(int k=0;k<D;++k)
            {
                if(!ext[nxt[j][k]]) dp[i][nxt[j][k]] = (dp[i][nxt[j][k]]+dp[i-1]
[j])%M;
            }
        }
    }
    int ans = 1;
    for(int i=1;i<=m;++i)   ans = (ans*26)%M;
    for(int i=0;i<=idx;++i) ans = (ans-dp[m][i]+M)%M;
    cout << ans << '\n';
}
```

模式串互相匹配

求模式串在模式串的出现次数

等同于求有多少个属于Y的节点的fail指针直接或间接指向X的结束位置

那么我们对fail求dfn，然后跑**trie树**，离线询问并用树状数组维护出现次数

> 因为是模式串在模式串，所以只能跑原始trie树

```
int m,idx,tot;
int dfn[N],sz[N],t[N];
int nxt[N][D],trie[N][D],fail[N],ans[N],mp[N],fa[N];
vector<int> ft[N];
vector<PII> eq[N];
string st,s0;

void put(int p,int z)
{
    for(int i=p;i<=tot;i+=i&-i) t[i] += z;
}

int query(int l,int r)
{
    int res = 0;
    for(int i=r;i>=1;i-=i&-i)   res += t[i];
    for(int i=l-1;i>=1;i-=i&-i) res -= t[i];
    return res;
}

void dfs(int p)
{
    dfn[p] = ++ tot;
    sz[p] = 1;
    for(auto &i : ft[p])
    {
        dfs(i);
        sz[p] += sz[i];
    }
}

void build_acam(void)
{
    queue<int> q;
    for(int i=0;i<D;++i)
    {
        if(nxt[0][i])
        {
            q.push(nxt[0][i]);
            ft[0].push_back(nxt[0][i]);
        }
    }
    while(q.size())
    {
        int p = q.front();  q.pop();
        for(int i=0;i<D;++i)
```

```
        {
            if(!nxt[p][i])      nxt[p][i] = nxt[fail[p]][i];
            else
            {
                int ps = nxt[p][i]; q.push(ps);
                fail[ps] = nxt[fail[p]][i];
                ft[fail[ps]].push_back(ps);
            }
        }
    }
    dfs(0);
}

void dfs_trie(int p)
{
    put(dfn[p],1);
    for(auto &[x,id] : eq[p])   ans[id] = query(dfn[x],dfn[x]+sz[x]-1);
    for(int i=0;i<26;++i)
    {
        if(trie[p][i])  dfs_trie(trie[p][i]);
    }
    put(dfn[p],-1);
}

signed main(void)
{
    Start;
    string s0;  cin >> s0;
    int cnt = 0,p=0;
    for(auto &i : s0)
    {
        if(i == 'B')
        {
            if(st.size())   st.erase(st.end()-1);
            p = fa[p];
        }
        else if(i == 'P')
        {
            for(auto &j : st)
            {
                int c = j-'a';
                if(!nxt[p][c])  nxt[p][c] = ++ idx;
                trie[p][c] = nxt[p][c];
                fa[nxt[p][c]] = p;
                p = nxt[p][c];
            }
            mp[++ cnt] = p;
            st.clear();
        }
        else    st += i;
    }
    cin >> m;
    build_acam();
    for(int i=1;i<=m;++i)
    {
        int x,y;    cin >> x >> y;
```

```
        eq[mp[y]].push_back({mp[x],i});
    }
    dfs_trie(0);
    for(int i=1;i<=m;++i)   cout << ans[i] << '\n';
    return 0;
}
```