2022 香农先修班第 12 次课题解

游戏人生第二季定档

心算即可：

```
print('''176
and
a
''')
```

也可以代码实现，展示 C 风格字符串和 `std::string` 两种代码：

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define sc(x) scanf("%lld", &x)
char s[] = "never gonna give you up, never gonna let you down, never gonna run around and desert you, never gonna make you cry, never gonna say goodbye, never gonna tell a lie and hurt you", r[30], prefix[30], suffix[30];
set<string> t;
decltype(strchr(s, ' ')) pos, pre = s; //即char* pos
signed main()
{
    printf("%d\n", strlen(s));
    const char *sep = ", ";   //必须const
    char *u = strtok(s, sep); // strtok用法模板
    while (u)
    {
        t.insert(u);
        u = strtok(nullptr, sep);
    }
    printf("%s\n", (++t.begin())->c_str());
    for (auto i : t)
    {
        strcpy(r, i.c_str());
        bool isPali = true;
        for (int i = 0, j = strlen(r) - 1; i < j; ++i, --j) //回文串模板
        {
            if (r[i] != r[j])
            {
                isPali = false;
                break;
            }
        }
        if (isPali)
        {
            printf("%s ", r);
        }
    }
    printf("\n");
    for (auto i : t)
```

```
    {
        strcpy(r, i.c_str());
        for (int i = 1, n = strlen(r); i < n; ++i) //真前/后缀所以不取等
        {
            strncpy(prefix, r, i);
            strncpy(suffix, r + n - i, i);
            if (strncmp(prefix, suffix, i) == 0) //判前后缀相等模板
            {
                printf("%s ", r);
                break;
            }
        }
    }
    return 0;
}
```

```
#include <bits/stdc++.h>
using namespace std;
string s = "never gonna give you up, never gonna let you down, never gonna run around and desert you, never gonna make you cry, never gonna say goodbye, never gonna tell a lie and hurt you";
set<string> t;
signed main()
{
    cout << s.size() << '\n';
    s += ' ';
    int pos = 0, prev = 0;
    replace(s.begin(), s.end(), ',', ' ');
    while ((pos = s.find(' ', prev)) != -1) //分割字符串模板
    {
        string sub = s.substr(prev, pos - prev);
        if (sub.size())
            t.insert(sub);
        prev = pos + 1;
    }
    cout << (*(++t.begin())) << '\n';
    for (auto i : t)
    {
        string j = i;
        reverse(j.begin(), j.end());
        if (j == i)
            cout << j << ' ';
    }
    cout << '\n';
    for (auto u : t)
        for (int i = 1; i < u.size(); ++i)
            if (u.substr(0, i) == u.substr(u.size() - i, i))
                cout << u << ' ';
    return 0;
}
```

字符串匹配

模板题。解析参考课件。下面给出字符串哈希和KMP两种代码。

字符串哈希：

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef unsigned long long ull;
#define mn 2000010
char s[mn], t[mn];
ull p = 131, pw[mn], h[mn], ht, n, m, cnt;
signed main()
{
    pw[0] = 1;
    for (ll i = 1; i < mn; ++i)
    {
        pw[i] = pw[i - 1] * p;
    }
    scanf("%s%s", s + 1, t + 1);
    n = strlen(s + 1), m = strlen(t + 1);
    for (ull i = 1; i <= n; ++i)
    {
        h[i] = h[i - 1] * p + s[i];
    }
    for (ull i = 1; i <= m; ++i)
    {
        ht = ht * p + t[i];
    }
    for (ull lf = 1, rf = m; rf <= n; ++lf, ++rf)
    {
        if (h[rf] - h[lf - 1] * pw[rf - lf + 1] == ht)
        {
            printf("%lld ", lf), ++cnt;
        }
    }
    if (!cnt)
    {
        printf("-1");
    }
    return 0;
}
```

KMP：

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define mn 2000010
char s[mn], t[mn];
ll kmp[mn], ns, nt, cnt;
signed main()
{
    scanf("%s%s", s + 1, t + 1);
    ns = strlen(s + 1), nt = strlen(t + 1);
    for (ll i = 2, j = 0; i <= nt; ++i)
    {
        while (j > 0 && t[j + 1] != t[i])
        {
            j = kmp[j];
```

```
        }
        if (t[j + 1] == t[i])
        {
            ++j;
        }
        kmp[i] = j;
    }
    for (ll i = 1, j = 0; i <= ns; ++i)
    {
        while (j > 0 && t[j + 1] != s[i])
        {
            j = kmp[j];
        }
        if (t[j + 1] == s[i])
        {
            ++j;
        }
        if (j == nt)
        {
            ++cnt;
            printf("%lld ", i - j + 1);
            j = kmp[j];
        }
    }
    if (cnt == 0)
    {
        printf("-1");
    }
    return 0;
}
```

随机生成的字符串岂不是随便搞都能过

所求即使得下面表达式成立的 (i,j) 数目：

$$a[1..i]+a[1..j]=b[1..i+j]$$

为简单起见，可以拆分为两个表达式，即：

$$\begin{cases}a[1..i]=b[1..i]\\a[1..j]=b[i+1..i+j]\end{cases}$$

这是因为根据字符串哈希的推论很容易求 $b[i+1..i+j]$，但是求 $a[1..i]+a[1..j]$ 的哈希值不那么直观。(具体怎么求下文介绍)

转化为字符串哈希表达式，所求即：

$$\begin{cases}h_a(i)=h_b(i)\\h_a(j)=h_b(j)-p^{i+j-(i+1)+1}h_b(i+1-1)=h_b(j)-p^jh_b(i)\end{cases}$$

我们发现，对任意字符串 S,T，若 $S=T$，则 $|S|=|T|$ 且恒有：

$$\forall 1\le i\le |S|,S[1..i]=T[1..i]$$

说人话就是若两字符串相等，则这两字符串相同长度的前缀分别相等

转化为这道题而言，那意味着当找到最大的 j 满足题意时，$\forall 1 \leq k \leq j$ 均满足题意，且 $\forall k > j$ 均不满足题意。这形成了单调性，因此可以二分 $j(1 \leq j \leq |a|)$，找到最大的满足题意的 j，然后将 cnt 累加 j (即长为 j 的区间 $j' \in [1, j]$ 都满足题意)。二分时每次哈希复杂度为 $O(1)$，二分复杂度为 $O(|a|)$。

时间复杂度为 $O(|b| + |a| \log |a|)$

参考代码：

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef unsigned long long ull;
#define mn 100010
ull na, nb, p = 131, pw[mn], ha[mn], hb[mn], cnt;
char a[mn], b[mn];
signed main()
{
    pw[0] = 1;
    for (ull i = 1; i < mn; ++i)
    {
        pw[i] = pw[i - 1] * p;
    }
    scanf("%s%s", a + 1, b + 1);
    na = strlen(a + 1), nb = strlen(b + 1);
    for (ull i = 1; i <= na; ++i)
    {
        ha[i] = ha[i - 1] * p + a[i];
    }
    for (ull i = 1; i <= nb; ++i)
    {
        hb[i] = hb[i - 1] * p + b[i];
    }
    for (ull i = 1; i <= na; ++i)
    {
        if (ha[i] != hb[i])
        {
            continue;
        }
        ull lf = 1, rf = na, cf, j = 0;
        while (lf <= rf)
        {
            cf = (lf + rf) >> 1;
            if (ha[cf] == hb[i + cf] - hb[i] * pw[cf])
            {
                lf = cf + 1, j = cf;
            }
            else
            {
                rf = cf - 1;
            }
        }
        cnt += j;
    }
    printf("%lld", cnt);
    return 0;
}
```

附：若想要求 $a[1..i]+a[1..j]$，则串长为 $i+j$，其哈希表达式为：

$$\begin{aligned}&(\sum_{k=1}^{i}a_kp^{i+j-k})+(\sum_{k=1}^{j}a_kp^{i+j-i-k})\\=&p^j(\sum_{k=1}^{i}a_kp^{i-k})+(\sum_{k=1}^{j}a_kp^{j-k})\\=&p^jh(i)+h(j)\end{aligned}$$

并不是朴素认知的 $h(i)+p^jh(j)$

用这种解法，只需要将二分时的 `if` 改成如下表达式即可：

```
if (pw[cf] * ha[i] + ha[cf] == hb[i + cf])
```

字符串模糊匹配

一种解法是字符串哈希+二分。

允许有 k 个不一样，那么可以从当前位置开始找 k 次，对每次找的过程，用二分法贪心找到最长完全匹配的位置，然后把这个位置的下一个位置字符跳过(用掉一次模糊的机会)，然后继续下一次找的过程，如果这个过程里匹配完了 T，那么模糊匹配成功，输出这个位置；如果 k 次跳过都用完了，然后再找(最后找已经不能跳了)，还没匹配完，那么匹配失败。

简单总结就是：k 次失配的模糊匹配等于 $k+1$ 次准确匹配(前 k 次准确贪心匹配每次完毕后跳过一个失配字符)。

因为对每个位置需要匹配 $k+1$ 次，每次都是用二分法(二分长度是 $|T|$)，所以时间复杂度为 $O(|S|+|T|+k|S|\log|T|)$

佬们也可以试试 FFT，我就不放 FFT 的代码了(逃

具体实现细节见代码：

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef unsigned long long ull;
#define mn 1000010
char s[mn], t[mn];
ull p = 131, pw[mn], ha[mn], hb[mn], ht, cnt, k;
signed main()
{
    pw[0] = 1;
    for (ll i = 1; i < mn; ++i)
    {
        pw[i] = pw[i - 1] * p;
    }
    scanf("%llu%s%s", &k, s + 1, t + 1);
    ll n = strlen(s + 1), m = strlen(t + 1);
    for (ll i = 1; i <= n; ++i)
    {
        ha[i] = ha[i - 1] * p + s[i];
    }
    for (ll i = 1; i <= m; ++i)
```

```
    {
        hb[i] = hb[i - 1] * p + t[i];
    }
    for (ll i = 1; i <= n - m + 1; ++i) //小心n-m+1负数
    {
        ll ok = 0, prea = i - 1, preb = 0;
        for (ull g = 0; g <= k;)
        {
            ll lf = 1, rf = m - preb, cf, res = -1;
            while (lf <= rf)
            {
                cf = (lf + rf) >> 1;
                ull va = ha[prea + cf] - ha[prea] * pw[cf];
                ull vb = hb[preb + cf] - hb[preb] * pw[cf];
                if (vb == va)
                {
                    res = cf, lf = cf + 1;
                }
                else
                {
                    rf = cf - 1;
                }
            }

            if (res != -1)
            {
                prea += res, preb += res;
            }
            if (preb >= m)
            {
                ok = 1;
                break;
            }
            ++prea, ++preb, ++g;
            if (g <= k && preb >= m)
            {
                ok = 1;
                break;
            }
        }
        if (ok)
        {
            printf("%lld ", i), ++cnt;
        }
    }
    if (!cnt)
    {
        printf("-1");
    }
    return 0;
}
```

最长公共子串

字符串哈希+二分。

二分最长长度 cf，然后对当前最长长度，对每个字符串 S_i，枚举所有长为 cf 的哈希值。开 `set` 或 `unordered_set` 作为集合存哈希值，第 i 个集合代表前 i 个字符串的公共哈希值，如果在前 $i-1$ 个字符串的并集里出现过这个哈希值，那么将其加入到前 i 个字符串的并集中。特别地，$i=1$ 时无条件加入。如果第 n 个集合有哈希值，证明当前长度 cf 可行，可继续向右逼近；否则向左逼近。

为了节省内存，可以把 n 个集合并集压缩成只需要两个并集，即当前并集和上一个并集，可以节省大量空间。(事实表明省一倍)

使用 `unordered_set` 比 `set` 快三到四倍。

其他实现细节，如动态数组，见代码。

二分长度为 cf 时，遍历所需复杂度为 $O(\sum(|S_i|-cf+1)=\sum|S|)$，遍历次数不超过 $\log\sum|S|$ 次，所以时间复杂度为 $O(|S|\log\sum|S|)$

参考代码：

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef unsigned long long ull;
#define sc(x) scanf("%lld", &x)
#define mn 1000010
ull pw[mn], p = 131;
ll n, lf, rf = 1e9, cf, ans;
set<ull> m[2]; //前i个串的最长公共子串，压缩数组n为2
string s[mn];
vector<ull> h[mn];
signed main()
{
    cin.tie(0)->ios::sync_with_stdio(0);
    pw[0] = 1;
    for (ll i = 1; i < mn; ++i)
    {
        pw[i] = pw[i - 1] * p;
    }
    cin >> n;
    for (ll i = 1; i <= n; ++i)
    {
        cin >> s[i];
        rf = min(rf, (ll)s[i].size());
        h[i].emplace_back(0);
        for (ll j = 1, je = s[i].size(); j <= je; ++j)
        {
            h[i].emplace_back(h[i].back() * p + s[i][j - 1]);
        }
    }
    while (lf <= rf)
    {
        cf = (lf + rf) >> 1;
        bool ok = true;
        for (ll i = 1, now = 1, pre = 0; i <= n; ++i, now ^= 1, pre ^= 1)
        {
            m[now].clear();
            for (ll j = 1, je = s[i].size() - cf + 1; j <= je; ++j)
            {
                ull v = h[i][j + cf - 1] - h[i][j - 1] * pw[cf];
                if (i == 1 || m[pre].find(v) != m[pre].end())
```

```
                {
                    m[now].insert(v);
                }
            }
            if (m[now].size() == 0)
            {
                ok = false;
                break;
            }
        }
        if (ok)
        {
            ans = cf, lf = cf + 1;
        }
        else
        {
            rf = cf - 1;
        }
    }
    printf("%lld", ans);
    return 0;
}
```

最短周期

若周期为 p，表明：$S[1..p]=S[p+1..2p]=\cdots=S[|S|-p,|S|]$，即有 $S[1..p]=S[|S|-p,|S|]$，且字符串可以按周期分为 $m=\dfrac{|S|}{p}$ 段。那么把前 $m-1$ 段拼接得 $S[1..|S|-p]$，后 $m-1$ 段拼接得 $S[p+1..S]$，那么由于 $A=A$ 所以 $(m-1)A=(m-1)A$，即 $S[1..|S|-p]=S[p+1..S]$。所以它们分别是 S 的最长真前缀和最长真后缀。(最长前缀和最长后缀是 S 本身)。那么根据前缀函数的定义，长度为 $\pi(|S|)$。根据减法，把 m 段长度($|S|$)减去 $m-1$ 段长度($\pi(|S|)$)就是一段的长度，即所求为 $|S|-\pi(|S|)$

使用 KMP 算法求前缀函数，时间复杂度为 $O(|S|)$

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define sc(x) scanf("%lld", &x)
#define mn 2000010
char s[mn];
ll kmp[mn], n;
signed main()
{
    scanf("%s", s + 1);
    n = strlen(s + 1);
    for (ll i = 2, j = 0; i <= n; ++i)
    {
        while (j && s[j + 1] != s[i])
        {
            j = kmp[j];
        }
        if (s[j + 1] == s[i])
        {
            ++j;
```

```
        }
        kmp[i] = j;
    }
    printf("%lld", n - kmp[n]);
    return 0;
}
```

前缀出现次数

首先毋容置疑，每个前缀至少出现一次，即出现在首位置 1 。

对于一个前缀函数 $\pi(i)$ ，若其不为 0 ，代表了一个真前缀与真后缀相等，也就是说长为 $\pi(i)$ 的真前缀在后面真后缀的位置出现了一次。然后我们递归地继续往下看，如果 $\pi(\pi(i))$ 不为 0 ，代表了又有一个长为 $\pi(\pi(i))$ 的真前缀在后面真后缀的位置出现了一次.....不断如此递归，直到得到前缀函数值为 0 为止。

如果暴力用递归计算上面的式子，复杂度为 $O(|S|^2)$ (考虑 $S=$ `aaa...a` ，全是同一个字符)，所以下面我们要优化这个计算。

可以用一种累积的思想。记 $a[x]$ 代表长为 x 的前缀当前已知累积出现过了 $a[x]$ 次。初始值显然是 $\forall 1\le i\le n, a[i]=1$ 。然后我们发现前缀函数有性质 $\pi(x)<x$ ，也就是不会影响大于等于 x 的位置，所以我们可以从 $|S|$ 倒序遍历，先处理未被影响过的位置。当遍历到 i 时，若 $\pi(i)>0$ ，那么我们可以把 $a[i]$ 累加到 $a[\pi(i)]$ 去，代表已知 $a[\pi(i)]$ 多出现了 $a[i]$ 次，那么当以后遍历到 $\pi(i)$ 时，我们便把之前的累加和当前的情况一并处理了。也就是说如果长为 i 的前缀出现了 $a[i]$ 次，发现 $\pi(i)>0$ 时，因为真前缀等于真后缀，长为 $\pi(i)$ 的真后缀出现了 $a[i]$ 次，所以长为 $\pi(i)$ 的真前缀也出现过 $a[i]$ 次。

> 如果不能理解上述过程可以拿样例来动手手算一下

由于倒序遍历一次是 $O(|S|)$ 的，故时间复杂度为 $O(|S|)$ 。

参考代码：

```
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;
typedef long long ll;
#define sc(x) scanf("%lld", &x)
#define mn 2000010
char s[mn];
ll kmp[mn], n, ans[mn], res;
signed main()
{
    scanf("%s", s + 1);
    n = strlen(s + 1);
    for (ll i = 2, j = 0; i <= n; ++i)
    {
        while (j && s[j + 1] != s[i])
        {
            j = kmp[j];
        }
        if (s[j + 1] == s[i])
        {
            ++j;
        }
```

```
        kmp[i] = j;
    }
    fill(ans + 1, ans + 1 + n, 1);
    for (ll i = n; i >= 1; --i)
    {
        ans[kmp[i]] += ans[i];
    }
    for (ll i = 1; i < n; ++i)
    {
        res += i * ans[i];
    }
    printf("%lld", res);
    return 0;
}
```

前缀出现次数2

> 本题建立在 `前缀出现次数` 的基础上

为了能够利用上题的思路，可以构造一个字符串 $A = S + $ `#` $+ T$ (`#` 是任何不在题目字符集内的字符，为了防止周期混乱；如果题目有 ASCII 码所有字符集，可以尝试用 `int` 存字符集并设一个大于 `char` 的值作字符)，因为存在一个不在 T 内的字符，所以可以保证对任意 i， $\pi(i) \le |S|$ 。

对 A 求前缀函数。对 $a[i]$ ，$i \le |S|$ 时，含义是 S 长为 i 的前缀的在 T 的出现次数。且初始化 $\forall 1 \le i \le |S|, a[i] = 0$ 。对 $2 + |S| \le i \le 1 + |S| + |T|$ 即字符串 T 部分，初始化 $a[i] = 1$ ，其含义为 A 长为 i 的前缀的在 A 的出现次数。我们可以跟上题一样倒序积累，最后计算 $\forall 1 \le i \le |S|$ 范围内即可。因为 $\pi(i) \le |S|$ ，所以每次积累时累加到的值一定是累加到 $i \le |S|$ 内的，即直到最后，仍然对 $2 + |S| \le i \le 1 + |S| + |T|$ 恒有 $a[i] = 1$ 。

> 如果不能理解上述过程可以拿样例来动手手算一下

时间复杂度为 $O(|S| + |T|)$

参考代码：

```
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;
typedef long long ll;
#define sc(x) scanf("%lld", &x)
#define mn 4000010
char s[mn];
ll kmp[mn], n, m, ans[mn], res;
signed main()
{
    scanf("%s", s + 1); // S: s[1..n]
    n = strlen(s + 1);
    s[n + 1] = '#';
    scanf("%s", s + 2 + n); // T: s[n+2..n+2+m]
    m = strlen(s + 2 + n);
    for (ll i = 2, j = 0; i <= n + m + 1; ++i)
    {
        while (j && s[j + 1] != s[i])
        {
            j = kmp[j];
```

```
        }
        if (s[j + 1] == s[i])
        {
            ++j;
        }
        kmp[i] = j;
    }
    fill(ans + n + 2, ans + 2 + n + m, 1); //[n+2,n+m+2)
    for (ll i = n + m + 1; i >= 1; --i)
    {
        ans[kmp[i]] += ans[i];
    }
    for (ll i = 1; i <= n; ++i)
    {
        res += i * ans[i];
    }
    printf("%lld", res);
    return 0;
}
```

字典树

模板题。具体参见课件。

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define sc(x) scanf("%lld", &x)
#define mn 500010
#define cs 26 // charse length
char s[mn];
ll n, m, ns, cnt = 1, t[mn][cs], w[mn], ans, p;
signed main()
{
    for (sc(n); n--;)
    {
        scanf("%s", s + 1);
        ns = strlen(s + 1), p = 1;
        for (ll j = 1, si; j <= ns; ++j)
        {
            ++w[p];
            si = s[j] - 'a';
            if (t[p][si] == 0)
            {
                t[p][si] = ++cnt;
            }
            p = t[p][si];
        }
        if (p)
        {
            ++w[p];
        }
    }
    for (sc(m); m--;)
    {
```

```
        scanf("%s", s + 1);
        ns = strlen(s + 1), p = 1;
        for (ll j = 1, si; j <= ns; ++j)
        {
            si = s[j] - 'a';
            p = t[p][si];
        }
        printf("%lld\n", w[p]);
    }
    return 0;
}
```

Browser Games

题意核心内容翻译：有 n 个字符串，对每个 i ，求最少的前缀数量，使得通过这些前缀只能找到前 i 个字符串，不能找到剩余的字符串。保证字符串间互不为前缀。

建树时可以先对每个节点加 1 ，那么点权代表从根节点到当前节点形成的前缀可以匹配到多少个字符串。初始时每个字符串都要求不能被匹配到，所以点权实质代表当前前缀不应匹配到的字符串有多少个。

然后顺次遍历 i 个字符串，每次遍历时：对第 i 个字符串的每个字符，在树上将其点权减一，代表第 i 个字符串移除出不应匹配的字符串，如果减 一后点权为 0 ，代表当前前缀已经不能再匹配到任何不应匹配的字符串了，那么可以将当前前缀作为题目要求的 `confirmation prefix` ，答案加一。

特别注意的是，当前前缀 p 加入后，以 p 为前缀的所有本来被加入过的前缀都应该被删除，以达到“最少”的目的，因为它们已经冗余了。

以 `abc` ，`abd` ，`abef` ，`abeg` ，`ba` 为例。 $i=1$ 时找到的当前前缀是 `abc` ，而 `i=2` 找到的是 `abd` ，`i=3` 时找到的是 `abef` ，但 `i=4` 时找的是 `a` ，这意味着前三个找到前缀都应该被删掉，因为 `a` 就能符合题意了，不需要多余的前缀。

为了实现这样的功能，我们可以在每次加入一个当前前缀 p 时，把字典树上 p 的所有前缀 (也就是 p 的所有祖先)都额外标记一个数值 $dels$，代表它成为前缀时，要额外删掉之前已经存在的 `confirmation prefix` 的数目。然后每次新增一个当前前缀 p 时，先加 1 代表选中这个前缀加入答案，然后再删掉标记的数目 $dels$，代表删掉多少个已经存在的 `confirmation prefix` 。

这个额外标记的数值具体而言，就是当前前缀 p 对答案的贡献值 $1-dels$ ，而并不是简单地标记加一。这是因为如果当前前缀 p 加入后删掉了其子树上的全部前缀(即之前已经存在的 `confirmation prefix`)，那么这个删除对 p 的所有祖先都是生效的，即对这些祖先的子树也删掉了。所以也要减去 $dels$ 。

时间复杂度是 $O(n|S|)$ ，空间复杂度是 $O(28n|S|)$

> 本题还有强化版本，限制空间为 $32\ MB$ ，见2021牛客多校第十场，感兴趣可以自行尝试，解题思路见下文。

参考代码：

```
#include <bits/stdc++.h>
using namespace std;
typedef int ll;    // not long long
#define mn 2500010 // 5e4*50
ll n, dels[mn], t[mn][28], cnt = 1, w[mn], p, ns, si, stak[55], ans;
char s[50010][55];
ll f(char c)
```

```
{
    return c == '.' ? 26 : (c == '/' ? 27 : c - 'a');
}
signed main()
{
    scanf("%d", &n);
    for (ll i = 1; i <= n; ++i)
    {
        scanf("%s", s[i] + 1);
        p = 1;
        ns = strlen(s[i] + 1);
        for (ll j = 1; j <= ns; ++j)
        {
            si = f(s[i][j]);
            if (t[p][si] == 0)
            {
                t[p][si] = ++cnt;
            }
            p = t[p][si];
            ++w[p];
        }
    }
    for (ll i = 1; i <= n; ++i)
    {
        p = 1;
        ns = strlen(s[i] + 1);
        for (ll j = 1; j <= ns; ++j)
        {
            si = f(s[i][j]);
            p = t[p][si];
            if (--w[p] == 0)
            {
                ll dt = 1 - dels[p];
                ans += dt;
                for (ll k = 1; k < j; ++k)
                {
                    dels[stak[k]] += dt;
                }
                break;
            }
            stak[j] = p;
        }
        printf("%d\n", ans);
    }
    return 0;
}
```

拓展阅读——空间复杂度 $O(n)$ 的解法：

只存关键节点，即叶子节点的 LCA ，有点像虚树的思维，可以得知加上叶子节点最多有 $O(2n)$ 个点。这些多出来的非叶子节点就是叶子的 LCA 及 LCA 的 LCA 。

因为我们不可能先建出树(不能真的建树然后建虚树)，会 MLE ，所以可以用别的方法。可以用一种基数排序的思想，从第一个字符开始作关键字，进行排序。设当前以第 d 个字符作关键字，排序后，可以 $O(n)$ 比较每个字符串第 d 个字符是否一样，如果第 d 位存在不重复的字符(即所有字符串只有一个字符串第 d 位取这个字符)，一个长为 d 的前缀可以作为增加的节点，并把它作为叶子节点(而不是这个字符串作叶子节点)。否则，就是存在重复的，设有 m 个第 d 位为同一个值的字符串，说明当前一定可以继

续划分。那么所有第 d 位是该字符的字符串全部合在一个非叶子节点里，然后找到下一个使得这些选中字符串存在差异的第 d' 位，以这些选中的字符串为整体，长为 d' 的前缀(长为 $[d,d']$ 的都符合题意)作为一个新节点，其能匹配到 m 的字符串，并以第 d' 位为关键字，继续向下递归，执行上述过程。本质上就是分治的思维。(具体实现参见代码)

最坏情况下需要走 $|S|$ 层，每层不断二分范围是最坏情况，如果用快排实现基数排序的内层排序，那么一次排序复杂度为 $O(n\log n)$，那么可以复杂度递推式为：$T(n)=2T(\frac{n}{2})+n\log n$，所以：

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2})+n\log n \\
&= 4T(\frac{n}{4})+n\log n+2\frac{n}{2}\log\frac{n}{2} \\
&= 4T(\frac{n}{4})+n\log n+n(\log n-\log 2) \\
&= 4T(\frac{n}{4})+2n\log n \\
&= 4T(\frac{n}{8})+3n\log n \\
&= \cdots \\
&= \log n\cdot n\log n \\
&= n\log^2 n
\end{aligned}
$$

所以最坏复杂度为 $O(n\log^2 n)$。

事实上本题下面的过题代码用时在 SCNUOJ 的数据快了十倍，在牛客的过题情况如下 所示

参考代码：

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef int ll;
#define mn 100010
#define mt 200010
#define ms 102 //注意这道题|S|=100
ll n, cnt, id[mt], fa[mt], w[mt], dp[mt], ans;
char s[mn][ms];
void build(ll lf, ll rf, ll dep, ll rot)
{
    sort(id + lf, id + rf + 1, [&](ll x, ll y)
         { return s[x][dep] < s[y][dep]; }); //[&]的&支持外部变量
    for (ll l = lf, r = lf; r <= rf;)
    {
        if (r == rf || s[id[r]][dep] != s[id[r + 1]][dep])
        {
            if (l == r)
            {
                fa[id[r]] = rot;
                w[id[r]] = 1;
                ++l, ++r;
                continue;
            }
            ll skip = 1;
            while (true)
            {
                bool allsame = true;
                for (ll i = l; i < r; ++i)
                {
                    if (s[id[i]][dep + skip] != s[id[i + 1]][dep + skip])
```

```
                    {
                        allsame = false;
                        break;
                    }
                }
                if (allsame)
                {
                    ++skip;
                }
                if (!allsame)
                {
                    ++cnt;
                    fa[cnt] = rot;
                    w[cnt] = r - l + 1;
                    build(l, r, dep + skip, cnt);
                    l = r + 1;
                    break;
                }
            }
        }
        ++r;
    }
}
signed main()
{
    scanf("%d", &n), cnt = n;
    for (ll i = 1; i <= n; ++i)
    {
        scanf("%s", s[i] + 1);
        id[i] = i;
    }
    build(1, n, 1, 0);
    for (ll i = 1; i <= n; ++i)
    {
        ll no = i, p = -1;
        while (no)
        {
            if (--w[no] == 0)
            {
                p = no;
                ans -= dp[no];
            }
            no = fa[no];
        }
        assert(p != -1);
        ++dp[fa[p]];
        ++ans;
        printf("%d\n", ans);
    }
    return 0;
}
```

最长异或路径

路径异或可以划分为经过根节点的和不经过的。经过的可以以根为一端划分为两段。不经过的，可以增加上相同的经过根 P 的路径，即 $AB=AC+CB=AC+CP+PC+CB$(C 是 $lca(A,B)$，实际写代码时不必求出)，其中异或和 CP,PC 抵消掉了。所以可以预处理每点到根的异或和得到一条路径，然后枚举每个这样的路径，对这些路径建01-trie，从高位开始贪心地找可以异或的位，相当于贪心地找到另一半路径。

> 这个路径划分思路跟点分治有点异曲同工之妙，如果您学过点分治相信比较好理解

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef int ll;
#define sc(x) scanf("%d", &x)
#define mn 1000010
struct edge
{
    ll to, nx, w;
} e[mn << 1];
ll hd[mn], cnt, tot, n, u, v, w, sum[mn << 1], ans;
void adde(ll &u, ll &v, ll &w)
{
    e[++cnt] = {v, hd[u], w};
    hd[u] = cnt;
}
void dfs(ll u, ll fa)
{
    for (ll i = hd[u]; i; i = e[i].nx)
    {
        ll v = e[i].to, w = e[i].w;
        if (v != fa)
        {
            sum[v] = w ^ sum[u]; //子树边权前缀异或和
            dfs(v, u);
        }
    }
}
struct trie
{
    ll c[2];
} t[mn << 1];
void build(ll v, ll x)
{
    for (ll i = (1 << 30); i; i >>= 1)
    {
        bool b = v & i; //v&i暴毙 不用bool暴毙
        if (!t[x].c[b])
        {
            t[x].c[b] = ++tot;
        }
        x = t[x].c[b];
    }
}
ll query(ll v, ll x)
{
    ll res = 0;
    for (ll i = (1 << 30); i; i >>= 1)
    {
        bool b = v & i;
```

```
        if (t[x].c[!b])
        {
            res += i;
            x = t[x].c[!b];
        }
        else
        {
            x = t[x].c[b];
        }
    }
    return res;
}

signed main()
{
    sc(n);
    for (ll i = 1; i < n; ++i)
    {
        sc(u), sc(v), sc(w);
        adde(u, v, w), adde(v, u, w);
    }
    dfs(1, 0);
    for (ll i = 1; i <= n; ++i)
    {
        build(sum[i], 0);
    }
    for (ll i = 1; i <= n; ++i)
    {
        ans = max(ans, query(sum[i], 0));
    }
    printf("%d", ans);
    return 0;
}
```

最长回文子串

模板题。请参考课件。

```
#include <bits/stdc++.h>
using namespace std;
#define mn 22000010
char s[mn], ch;
int p[mn], n, ans, r, c;
signed main()
{
    s[0] = '#', s[++n] = '#';
    while (EOF != (ch = getchar()))
    {
        s[++n] = ch, s[++n] = '#';
    }
    for (int i = 1; i <= n; ++i)
    {
        if (i <= r)
        {
            p[i] = min(p[c * 2 - i], r - i + 1);
```

```
        }
        while (s[i + p[i]] == s[i - p[i]])
        {
            ++p[i];
        }
        if (i + p[i] > r)
        {
            r = p[i] + i - 1, c = i;
        }
        ans = max(ans, p[i]);
    }
    printf("%d", ans - 1);
    return 0;
}
```