



南京工業大學
NANJING TECH
UNIVERSITY

ICPC Template Manual



作者: 贺梦杰

October 11, 2019

Contents

1	字符串 L	2
1.1	Hash	3
1.1.1	基础 Hash 匹配	3
1.2	KMP	4
1.2.1	前缀函数	4
1.2.1.1	朴素算法	4
1.2.1.2	第一个优化	4
1.2.1.3	第二个优化	4
1.2.2	在线 KMP	5
1.2.3	统计每个前缀出现次数	5
1.2.3.1	单串统计	5
1.2.3.2	双串统计	6
1.2.4	统计一个字符串本质不同的子串的数目	6
1.2.5	字符串压缩	6
1.2.6	根据前缀函数构建一个自动机	7
1.2.7	Gray 字符串	7
1.2.8	UVA11022 String Factoring	9
1.2.9	UVA12467 Secret word	10
1.2.10	UVA11019 Matrix Matcher	10
1.2.11	cf808G Anthem of Berland	11

Chapter 1

字符串 L

1.1 Hash

Hash 的核心思想在于，暴力算法中，单次比较的时间太长了，应当如何才能缩短一些呢？

如果要求每次只能比较 $O(1)$ 个字符，应该怎样操作呢？

我们定义一个把 string 映射成 int 的函数 f ，这个 f 称为是 Hash 函数。

我们需要关注的是时间复杂度和 Hash 的准确率。

通常我们采用的是多项式 Hash 的方法，即

$$f(s) = \sum (s[i] * b^i) \pmod{M}$$

其中 b 与 M 互质，且 M 越大错误率越小。(单次匹配错误率 $\frac{1}{M}$ ， n 次匹配的错误率为 $\frac{n}{M}$)

1.1.1 基础 Hash 匹配

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4  typedef long long ll;
5  // b和M互质；M可以尽量取大、随机化。对于本问题，无需建立关于M的数组，所以M最大可以达到1<<31大小。
6  const ll N = 1e3 + 10, b = 131, M = 1 << 20;
7
8  char s[2][N]; // s[0]是待匹配串，s[1]是模式串。下标从1开始
9  ll len[2];    // 两串的长度
10 ll Exp[N];    // Exp[i]=b^i
11
12 // 返回s[l..r]的哈希值 s[l]*Exp[1]+s[l+1]*Exp[2]+..
13 inline ll Hash(char s[], ll l, ll r) {
14     ll i, ret = 0;
15     for (i = l; l + i - 1 <= r; i++)
16         ret = (ret + Exp[i] * s[l + i - 1]) % M;
17     return ret;
18 }
19
20 vector<ll> ans; // 匹配子串的开始下标
21 inline void match() {
22     ans.clear();
23     ll i;
24     ll h0 = Hash(s[0], 1, len[1]); // 初始时待匹配串对应模式串那部分子串的哈希值
25     ll h1 = Hash(s[1], 1, len[1]); // 初始时模式串的哈希值
26     for (i = 1; i <= len[0] - len[1] + 1; i++) {
27         // 若两哈希值一致，则认为匹配
28         if ((h0 - h1 * Exp[i - 1]) % M == 0)
29             ans.push_back(i);
30         h0 = (h0 - Exp[i] * s[0][i] + Exp[i + len[1]] * s[0][i + len[1]]) % M; // 模式串向后移动，
           // 对应h0也要改变
31     }
32 }
33
34 int main() {
35     ios::sync_with_stdio(0);
36     cin.tie(0);
37
38     ll i, j;
39
40     // 初始化
41     Exp[0] = 1;
42     for (i = 1; i < N; i++)
43         Exp[i] = Exp[i - 1] * b % M;
44
45     cin >> (s[0] + 1) >> (s[1] + 1);
46     len[0] = strlen(s[0] + 1), len[1] = strlen(s[1] + 1);
47
48     match();
49
50     return 0;
51 }

```

1.2 KMP

1.2.1 前缀函数

给定一个长度为 n 的字符串 s (假定下标从 1 开始), 其**前缀函数**被定义为一个长度为 n 的数组 π , 其中 $\pi[i]$ 为既是子串 $s[1 \dots i]$ 的前缀同时也是该子串的后缀的最长真前缀 (proper prefix) 长度。一个字符串的真前缀是其前缀但不等于该字符串自身。根据定义, $\pi[1] = 0$ 。

前缀函数的定义可用数学语言描述如下:

$$\pi[i] = \max_{k=0 \dots i-1} \{k : s[1 \dots k] = s[i-k+1 \dots i]\}$$

举例来说, 字符串 `abcbcd` 的前缀函数为 $[0, 0, 0, 1, 2, 3, 0]$, 字符串 `aabaaab` 的前缀函数为 $[0, 1, 0, 1, 2, 2, 3]$ 。

1.2.1.1 朴素算法

直接按定义计算前缀函数:

```
1 // 朴素法求前缀函数O(n^3), 下标从1开始
2 void prefix_func0(char t[], int n, int pi[]) {
3     int i, k;
4     for (i = 1; i <= n; i++) // 对每一个子串
5         for (k = 0; k < i; k++) // 枚举前缀后缀长度, 并判断是否相等
6             if (!strcmp(t + 1, t + i - k + 1, k))
7                 pi[i] = k;
8 }
9
```

1.2.1.2 第一个优化

第一个重要的事实是相邻的前缀函数值至多增加 1。(如不然, 会产生矛盾)

所以当移动到下一个位置时, 前缀函数要么增加 1, 要么不变或减少。实际上, 该事实已经允许我们将复杂度降至 $O(n^2)$ 。因为每一步中前缀函数至多增加 1, 因此在总的运行过程中, 前缀函数至多增加 n , 同时也至多减小 n 。这意味着我们仅需进行 $O(n)$ 次字符串比较, 所以总复杂度为 $O(n^2)$ 。

```
1 void prefix_func1(char t[], int n, int pi[]) {
2     int i, j;
3     i = 2, j = 1;
4     while (i <= n) {
5         if (t[i] == t[j]) // 加1
6             pi[i] = pi[i - 1] + 1, ++j;
7         else { // 开始减
8             pi[i] = pi[i - 1];
9             while (strcmp(t + i - pi[i] + 1, t + 1, pi[i]))
10                 --pi[i];
11             j = pi[i] + 1;
12         }
13         ++i;
14     }
15 }
16
```

1.2.1.3 第二个优化

考虑计算位置 $i+1$ 的前缀函数 π 的值, 如果 $s[i+1] = s[\pi[i]+1]$, 显然 $\pi[i+1] = \pi[i] + 1$ 。

$$\underbrace{\overbrace{s_1 \ s_2 \ s_3}^{\pi[i]} \ \overbrace{s_4}^{s_4=s_{i+1}}}_{\pi[i+1]=\pi[i]+1} \ \dots \ \underbrace{\overbrace{s_{i-2} \ s_{i-1} \ s_i}^{\pi[i]} \ \overbrace{s_{i+1}}^{s_4=s_{i+1}}}_{\pi[i+1]=\pi[i]+1}$$

如果不是上述情况, 即 $s[i+1] \neq s[\pi[i]+1]$, 我们需要尝试更短的字符串。为了加速, 我们希望直接移动到最长的长度 $j < \pi[i]$, 使得在位置 i 的前缀性质仍得以保持, 也即 $s[1 \dots j] = s[i-j+1 \dots i]$:

$$\underbrace{\overbrace{s_1 \ s_2}^{\pi[i]} \ s_3 \ s_4}_{j} \ \dots \ \underbrace{\overbrace{s_{i-3} \ s_{i-2} \ s_{i-1} \ s_i}^{\pi[i]}}_{j} \ s_{i+1}$$

实际上, 如果我们找到了这样的 j , 我们仅需要再次比较 $s[i+1]$ 和 $s[j+1]$ 。如果它们相等, 则 $\pi[i+1] = j+1$, 否则, 我们就需要找小于 j 的最大的新的 j 使得前缀性质仍然保持, 如此反复, 直到 $s[i+1] = s[j+1]$ 或者确实完全找不到 (令 $j = -1$)。最后 $\pi[i+1] = j+1$ 。

所以我们已经有了一个大致框架, 现在仅剩的问题是对于满足 $s[1 \dots j] = s[i-j+1 \dots i]$ 的 j , 如何快速找到小于 j 的最大的新的 j , 我们令新的 j 为 k , 使得 $s[1 \dots k] = s[i-k+1 \dots i]$ 仍然满足。

$$\underbrace{s_1 s_2 s_3 s_4}_{k} \dots \underbrace{s_{i-3} s_{i-2} s_{i-1} s_i}_{k} s_{i+1}$$

由上图, 我们要求的是比 j 小的最大的 k , 而两边长度为 j 的前后缀本身是相等的, 那么新的长为 k 的前后缀则可以只放到最左边长为 j 的前缀中去考虑:

$$\underbrace{s_1 s_2 s_3 s_4}_{k}$$

即 $k = \pi[j]$, 而 $\pi[j]$ 之前已经求过了。

```

1 void prefix_func2(char t[], int n, int pi[]) {
2     int i, j;
3     pi[0] = -1, pi[1] = 0; // 确实没有找到任何相等的
4     for (i = 1; i < n; ++i) {
5         j = pi[i];
6         while (j >= 0 && t[j+1] != t[i+1]) // 若不相等, 找更小的新的j
7             j = pi[j];
8         pi[i+1] = j+1; // 最后得出pi[i+1]
9     }
10 }
11
12
```

1.2.2 在线 KMP

最基础的字符串匹配。下面的算法不是在线的, 但只要稍作修改就可以变成在线的了。

假设当前在 s 串的 i 处, cur 是当前 s 和 t 匹配的最大长度, 也就是 t 的长为 cur 的前缀和 s 中以 $s[i]$ 为右端点的子串相等

```

1 // 长为n的待匹配串s, 长为m的模式串t, 返回t在s中出现的次数
2 int kmp(char s[], int n, char t[], int m, int pi[]) {
3     int cur = 0, i, j, cnt = 0; // cur是当前pi值
4     for (i = 0; i < n; i++) {
5         j = cur;
6         while (j >= 0 && s[i+1] != t[j+1])
7             j = pi[j];
8         cur = j+1;
9         if (cur == m)
10             ++cnt;
11     }
12     return cnt;
13 }
14
```

1.2.3 统计每个前缀出现次数

1.2.3.1 单串统计

统计 s 的每个前缀在 s 中出现的次数。首先我们明确, 一个长度为 i 的前缀中会出现长度为 $\pi[i]$ 的前缀, 然后长度为 $\pi[i]$ 的前缀又会出现长度为 $\pi[\pi[i]]$ 的前缀, 等等。所以我们考虑后缀和的思想, 首先统计每一个位置的 $\pi[i]$, 然后将 $ans[i]$ 累加给长为 $ans[\pi[i]]$ 的前缀个数, 按照长度递减的顺序依次累加下去就可以了。最后再统计原始前缀, 即对每个 $ans[i]$ 加一。

```

1 // 统计s[]的每个前缀在s[]中出现的次数
2 inline void count_prefix(char s[], int n, int pi[]) {
3     prefix_func(s, n, pi); // 计算前缀函数

```

```

4     int i;
5     for (i = 1; i <= n; i++)
6         ++ans[pi[i]];
7     // 令真前后缀长度为i, 其个数为ans[i], 则长为i的真前后缀的真前后缀长为pi[i], 其原本个数为ans[pi[i]]
8     // 现在需要累加上它在更长的真前后缀中出现的次数。有点类似倍增
9     for (i = n; i >= 1; i--)
10        ans[pi[i]] += ans[i];
11    // 这里不能放到开头, 否则, 一开始ans[n]=1, 也就认为s有一个长为n的真前后缀
12    for (i = 1; i <= n; i++)
13        ++ans[i];
14 }
15

```

1.2.3.2 双串统计

给出串 s 和 t , 问 t 的每个前缀在 s 中出现的次数。首先运用类似 KMP 的思想, 通过 '#' 连接 t 和 s , 即 " $t\#s$ ", 设为 $link$, 对 $link$ 求前缀函数。接下来我们只关心与 s 有关的前缀函数值, 即 $i \geq m + 2$ 的 $\pi[i]$ 。

```

1 inline void count_prefix(char s[], int n, char t[], int m, int pi[]) {
2     // 连接字符串 t..#s..
3     strcpy(link + 1, t + 1);
4     link[m + 1] = '#';
5     strcpy(link + m + 2, s + 1);
6     // 计算前缀函数
7     prefix_func(link, n + m + 1, pi);
8
9     int i;
10    // 只关心#后面的pi值
11    for (i = m + 2; i <= n + m + 1; i++)
12        ++ans[pi[i]];
13    // pi[i]的值不会超过t的长度, 即i<=m
14    for (i = m; i >= 1; i--)
15        ans[pi[i]] += ans[i];
16 }
17

```

1.2.4 统计一个字符串本质不同的子串的数目

给定一个长度为 n 的字符串 s , 我们希望计算其本质不同子串的数目。

假设现在知道了当前 s 本质不同的子串的数目, 那么接下来可以考虑在原来的 s 末尾加上一个字符 c , 然后统计产生了多少新的子串。

我们枚举 i , 对每一个 i , 反转 $s[1 \dots i]$ 令其为 t , 然后对 t 求前缀函数, π_{max} 即为以 $s[i]$ 结尾的重复子串的数目, 那么 $|s| - \pi_{max}$ 即为以 $s[i]$ 结尾的新的子串的数目。

```

1 // 统计串s中本质不同的子串数目
2 inline int diff(char s[], int n) {
3     int i, ret = 0;
4     for (i = 1; i <= n; i++) {
5         reverse_copy(s + 1, s + 1 + i, t + 1); // 翻转s并存入t
6         int mx = prefix_func2(t, i, pi); // 略微修改一下prefix_func, 使其可以返回最大的pi值
7         ret += i - mx; // 统计新的子串数目
8     }
9     return ret;
10 }
11

```

1.2.5 字符串压缩

给定一个长度为 n 的字符串 s , 我们希望找到其最短的“压缩”表示, 也即我们希望寻找一个最短的字符串 t , 使得 s 可以被 t 的一份或多份拷贝的拼接表示。

显然, 我们只需要找到 t 的长度即可。知道了长度, 该问题的答案即为长度为该值的 s 的前缀。

直接说结论, 首先求 s 的前缀函数, 令 $k = n - \pi[n]$, 若 $n \bmod k = 0$, 则该长度为 k , 否则为 n 。

注: 上述情况为 s 恰好为最短循环节的倍数, 如果没有这个限制, 那么对于 $s[1 \dots i]$ 最短循环节长度永远是 $k = i - \pi[i]$ 。

```

1 // 计算s的最短压缩表示, 返回最短压缩的长度
2 int compress(char s[], int n, int pi[]) {
3     prefix_func(s, n, pi);
4     int k = n - pi[n];
5     if (n % k)
6         return n;
7     return k;
8 }
9

```

1.2.6 根据前缀函数构建一个自动机

让我们重新回到通过一个分割符将两个字符串拼接的新字符串。设模式串为 t , 待匹配串为 s , 则拼接成 $t + \# + s$ 。前面我们就知道, 我们只需要管 $t + \#$ 的前缀函数值就可以, 所以在这里我们的自动机也是如此。

自动机的状态为当前前缀函数的值, 而下一个读入自动机的字符则决定状态如何转移。下面我们就是要求状态转移表 aut , $aut[i][c]$ 表示当前前缀函数值为 i (也即当前处于 $t[i]$ 处) 下一个字符为 c 的转移的目标状态。

```

1 // 计算长度为n的字符串t[]的自动机的转移表, t[]下标从1开始
2 void compute_automaton0(char t[], int n, int pi[], int aut[][26]) {
3     prefix_func(t, n, pi); // 先求t[]的前缀函数
4     int i, j, c;
5     // 0是初始状态, 匹配长度为0; n是终止状态, 完全匹配
6     for (i = 0; i <= n; i++) {
7         // 转移的因素--下一个字符
8         for (c = 0; c < 26; c++) {
9             j = i;
10            // 通过不断跳转前缀来匹配下一个字符
11            while (j >= 0 && c + 'a' != t[j + 1])
12                j = pi[j];
13            aut[i][c] = j + 1;
14        }
15    }
16 }
17

```

在上面的代码中, 由于有 while 循环的存在, 所以时间复杂度为 $O(|\Sigma|n^2)$ 。

事实上, 我们可以通过动态规划来优化。我们注意到当下一个字符 $c \neq s[j + 1]$ 时, j 会跳转至 $\pi[j]$, 而在之前我们已经计算过所有 $aut[\pi[j]][c], \forall c \in \Sigma$, 所以我们可以直接利用 $aut[\pi[i]][c]$ 。时间复杂度 $O(|\Sigma|n)$ 。

```

1 // 计算长度为n的字符串t[]的自动机的转移表, t[]下标从1开始
2 void compute_automaton1(char t[], int n, int pi[], int aut[][26]) {
3     prefix_func(t, n, pi); // 先求t[]的前缀函数
4     int i, c;
5     // 0是初始状态, 匹配长度为0; n是终止状态, 完全匹配
6     aut[0][t[1] - 'a'] = 1;
7     for (i = 1; i <= n; i++) {
8         // 转移的因素--下一个字符
9         for (c = 0; c < 26; c++) {
10            // 利用动态规划的思想来优化, 在上面j跳转至pi[j]时, aut[pi[j]][c]对于所有的c都被计算过了
11            if (c + 'a' == t[i + 1])
12                aut[i][c] = i + 1;
13            else
14                aut[i][c] = aut[pi[i]][c];
15        }
16    }
17 }
18

```

1.2.7 Gray 字符串

首先定义 Gray 字符串, 令 $g[0] = ""$ (空串), $g[1] = "1"$, 之后 $g[i] = g[i - 1] + i + g[i - 1]$, 加号为字符串拼接。接下来我们考虑这样一个问题 (与 OIWiki 不同的是数据范围作了修改): 给定长为 n ($n \leq 1000$) 的字符串 s ($1 \leq s[i] \leq n$), 一个整数 k ($k \leq 1000$), 要求 s 在 $g[k]$ 中出现的次数。

这题是 kmp 自动机 + dp。

显然, $g[k]$ 的长度非常大, 我们不可能去构造。但我们可以好好利用 Gray 字符串递归的性质, 即 $g[i] = g[i-1] + i + g[i-1]$ 。

对 s 构建一个自动机 $aut[i][j]$, 表示从当前状态 i 通过输入的 j 到达的状态。

假设当前自动机处于状态 i , 接下来要处理 $g[j]$, 我们可以分为 3 步:

1. 从状态 i 开始处理 $g[j-1]$, 自动机到达状态 $t1$
2. 从状态 $t1$ 开始处理 j , 自动机到达状态 $t2$
3. 从状态 $t2$ 开始处理 $g[j-1]$, 自动机到达状态 $t3$ 。

其中 $t3$ 即为目标状态。

显然, 如果每一步都老老实实做的话, 工作量并未减少。仔细观察发现我们可以建立一个 dp 状态, $G[i][j]$ 表示自动机从状态 i 开始处理 $g[j]$, 处理完成后自动机所处的状态。那么上面的 3 步就可以变为如下形式:

$$t1 = G[i][j-1]$$

$$t2 = aut[t1][j]$$

$$t3 = G[t2][j-1]$$

由于 $i \leq n$ 且 $j \leq k$ 复杂度 $O(nk)$ 。

如何计算答案呢? 我们只要在自动机转移的过程中看当前状态是否为 $|s|$ 状态 (和 s 完全匹配) 即可。但是我们上面都是一跳一大步, 可能有的 $|s|$ 状态就给跳过去了。所以我们再用一个 $K[i][j]$ 记录自动机从状态 i 开始处理 $g[j]$ 直到处理完成, 这个过程中几次达到状态 $|s|$ 。显然有:

$$K[i][j] = K[i][j-1] + (t2 == |s|) + K[t2][j-1]$$

初始时, $aut[i][j]$ 可全部求出, $G[i][0]=i$, $K[i][0]=0$, 因为 $g[0]$ 为空串, 自动机不会转移, $g[0]$ 也不会包含 s 。显然, 最后答案为 $K[0][k]$ 。

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4  const int N = 1e3 + 10;
5
6  int s[N];
7  int n, k;
8  int pi[N];
9  int aut[N][N]; // 自动机, aut[i][j]表示当前状态通过输入j得到的下一个状态
10 int G[N][N]; // G(i,j)表示自动机从状态i开始处理g[j], 处理完成后自动机的状态
11 int K[N][N]; // K(i,j)表示自动机从状态i开始处理g[j], 处理完成后s在g[j]中出现的次数
12
13 void prefix_func(int t[], int n, int pi[]) {
14     int i, j;
15     pi[0] = -1, pi[1] = 0; // 确实没有找到任何相等的
16     for (i = 1; i < n; ++i) {
17         j = pi[i];
18         while (j >= 0 && t[j+1] != t[i+1]) // 若不相等, 找更小的新的j
19             j = pi[j];
20         pi[i+1] = j+1; //最后得出pi[i+1]
21     }
22 }
23
24 // 计算长度为n的字符串t[]的自动机的转移表, t[]下标从1开始
25 void compute_automaton1(int t[], int n, int pi[], int aut[][N]) {
26     t[n+1] = -1; // 结束符, 结束符应是字母表中没有的符号
27     prefix_func(t, n, pi); // 先求t[]的前缀函数
28     int i, c;
29     // 0是初始状态, 匹配长度为0; n是终止状态, 完全匹配
30     aut[0][t[1]] = 1;
31     for (i = 1; i <= n; i++) {
32         // 转移的因素--下一个字符
33         for (c = 1; c <= n; c++) {
34             // 利用动态规划的思想来优化, 在上面j跳转至pi[j]时, aut[pi[j]][c]对于所有的c都已经被计算过了
35             if (c == t[i+1])
36                 aut[i][c] = i+1;
37             else
38                 aut[i][c] = aut[pi[i]][c];
39         }

```

```

40     }
41 }
42
43 int main() {
44     ios::sync_with_stdio(0);
45     cin.tie(0);
46
47     int i, j;
48     cin >> n >> k;
49     for (i = 1; i <= n; i++)
50         cin >> s[i];
51
52     compute_automaton1(s, n, pi, aut);
53
54     // 初始化, 从状态i开始处理g[0], 由于g[0]是空串, 所以自动机不会转移, s也不会出现在g[0]中
55     for (i = 0; i <= n; i++)
56         G[i][0] = i, K[i][0] = 0;
57     // 类似于动态规划的递推
58     for (j = 1; j <= k; j++) {
59         for (i = 0; i <= n; i++) {
60             int mid = aut[G[i][j - 1]][j];
61             G[i][j] = G[mid][j - 1];
62             K[i][j] = K[i][j - 1] + (n == mid) + K[mid][j - 1];
63         }
64     }
65
66     cout << K[0][k] << endl;
67
68     return 0;
69 }
70

```

1.2.8 UVA11022 String Factoring

给定一个字符串 $s(|s| \leq 80)$, 问最小压缩长度。例如, $AAA = (A)^3$ 最小长度为 1, $CABAB = C(AB)^2$ 最小长度为 3, 再如 $POPPPOP$ 既可以是 $PO(P)^2OP$ 也可以是 $(POP)^2$, 但后者长度为 3, 前者为 5, 所以后者更优。

这题是 kmp 字符串压缩 + 区间 dp。

$f[i][j]$ 表示 $s[i..j]$ 被压缩后的最短长度, $pi2[i][j]$ 表示以 i 为左端点, j 处的前缀函数值。先考虑最简单的区间 dp:

$$f(l, r) = \min_{l \leq k < r} \{f(l, m) + f(m + 1, r)\}$$

可以发现, 这是不完整的, 例如 ATTATT, 假设已知 $f(1, 3) = 2$ 和 $f(4, 6) = 2$, 接下来算 $f(1, 6)$ 就会等于 4, 而正确答案是 2。也就是说, 我们没有考虑 $s[l..r]$ 可能本身就可以被压缩。若 $s[l..r]$ 本身可以被压缩, 则一定存在一个最小循环节, 这我们可以通过上面的 $pi2[l][r]$ 来求。

若确实存在循环节, 设其长度为 k , 则 $f(l, r) = f(l, l + k - 1)$, 之所以这样, 是因为循环节本身也是可以被压缩的 (类似于问题)。之后再用上面的区间 dp, 就是对的了。

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4  const int N = 85;
5
6  char s[N];
7  int n;
8  int pi2[N][N], pi[N]; // pi2[i][j]表示以i为左端点, j处的前缀函数值
9  int f[N][N];          // f[i][j]表示s[i..j]被压缩后的最短长度
10
11 void prefix_func(char t[], int n, int pi[]) {
12     int i, j;
13     pi[0] = -1, pi[1] = 0; // 确实没有找到任何相等的
14     for (i = 1; i < n; ++i) {
15         j = pi[i];
16         while (j >= 0 && t[j + 1] != t[i + 1]) // 若不相等, 找更小的新的j
17             j = pi[j];
18     }
19 }

```

```

18     pi[i + 1] = j + 1; //最后得出pi[i+1]
19 }
20 }
21
22 int main() {
23     ios::sync_with_stdio(0);
24     cin.tie(0);
25
26     int i, j, k, l, r, m;
27
28     while (cin >> (s + 1)) {
29         if (s[1] == '*')
30             break;
31         n = strlen(s + 1);
32
33         // 求pi2[i][j]
34         for (i = 1; i <= n; i++)
35             prefix_func(s + i - 1, n - i + 1, pi2[i] + i - 1);
36
37         // dp求f(l,r), s[l..r]被压缩后的最短长度
38         for (i = 1; i <= n; i++) { // 枚举长度
39             for (l = 1; l + i - 1 <= n; l++) { // 枚举左端点
40                 r = l + i - 1; // 右端点
41                 f[l][r] = i; // 初始化为最坏情况
42                 int len = r - l + 1; // 当前区间的长度
43                 // 如果本身完全是循环节
44                 k = len - pi2[l][r];
45                 if (len % k == 0)
46                     f[l][r] = min(f[l][r], f[l][l + k - 1]); // 循环节的性质, 转化为已求的子问题
47                 // 区间DP
48                 for (m = l; m < r; m++)
49                     f[l][r] = min(f[l][r], f[l][m] + f[m + 1][r]);
50             }
51         }
52
53         cout << f[1][n] << endl;
54     }
55
56     return 0;
57 }
58

```

1.2.9 UVA12467 Secret word

给你一个串 S ($1 \leq |S| \leq 10^6$), 要求最长前缀, 这种前缀满足: 它的反转是 S 的子串。

前缀的反转是 S 的子串, 换句话说, 我们将 S 反转, 若它的某一个子串为原串 S 的前缀, 则该子串的反转就是满足条件的子串之一。

令 S' 为 S 的反转, 则 S 为模式串, S' 为待匹配串, 对 S 和 S' 用 kmp, 匹配过程中的最大 π 值即为答案。

1.2.10 UVA11019 Matrix Matcher

给定一个 $n * m$ 的大矩阵和一个 $x * y$ 的小矩阵, 问小矩阵在大矩阵中出现的次数。

令大矩阵为 $s[][]$, 小矩阵为 $t[][]$, 最暴力的想法:

设 $pi2[i][j]$ 为 $t[i]$ 在 j 处的前缀函数值。

设 $f(i, j, k)$ 为以 $t[i]$ 为模式串, $s[j]$ 为待匹配串, 匹配到 $s[j][k]$ 处的前缀函数值。

$pi2$ 可以直接通过 x 次 KMP 求出, f 可以通过枚举 (i, j) 对 $t[i]$ 和 $s[j]$ 用 KMP 求出。

最后枚举 t 在 s 中的左上角, 判断每一行匹配到最后的值是否为 y 即可。具体来说, 设当前 s 左上角为 (i, j) , 对 t 的第 k 行匹配, 若 $\forall k \in [1, x], f[k][i + k - 1][j + y - 1] = y$, 则匹配成功, 否则失败。

事实上, 这题正解应该是 **AC 自动机**。

1.2.11 cf808G Anthem of Berland

给出两个串 s 和 t , s 中有一部分是问号 (t 是确定的), 问号可以是任意字符, 问 t 在 s 中出现的最大次数。

最暴力的做法是直接 dfs 搜索, 时间爆炸。

由于每一个问号处都可能匹配到 t 的任意位置, 所以可以考虑 dp 优化, 设状态为 $f(i, j)$, 表示 $s[i]$ 处匹配到 $t[j]$ 处时 t 在 s 中出现的最大次数。如何转移呢? 当 $s[i]$ 处完全匹配 t 串, 此时 $f(i, j)$ 的值就要加 1, 否则不变。但是光这样还不够, 考虑 j 处的 $\pi[j]$ 值, 这意味着 t 的长度为 $\pi[j]$ 的前后缀是相等的, 也就是说值 $f(i, j)$ 也可以传递给 $f(i, \pi[j])$ 以及 $f(i, \pi[\pi[j]])$ 等等。

如果对每个 j 还要枚举 $\pi[j]$, 时间就会爆炸。所以我们还要借用前缀和的思想, $h(i, c)$ 表示当前 π 值为 i 且下一个字符为 c 的最优答案。我们按 j 从大到小处理, 在处理 j 时, 我们同时给 $h[\pi[j]][t[j+1]]$ 更新, 而当前的 $f(i, j)$ 也与 $h(j, s[i+1])$ 有关。