南京工業大學
NANJING TECH UNIVERSITY

ICPC Template Manual

作者: 贺梦杰

September 29, 2019

Contents

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

$$f(s) = \sum(s[i] * b^i)(\bmod M)$$

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

```cpp
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
// 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。
const ll N = 1e3 + 10, b = 131, M = 1 << 20;

char s[2][N];  // s[0]是待匹配串，s[1]是模式串。下标从1开始
ll len[2];     // 两串的长度
ll Exp[N];     // Exp[i]=b^i

// 返回s[l..r]的哈希值 s[l]*Exp[1]+s[l+1]*Exp[2]+..
inline ll Hash(char s[], ll l, ll r) {
    ll i, ret = 0;
    for (i = 1; l + i - 1 <= r; i++)
        ret = (ret + Exp[i] * s[l + i - 1]) % M;
    return ret;
}

vector<ll> ans;  // 匹配子串的开始下标
inline void match() {
    ans.clear();
    ll i;
    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
    ll h1 = Hash(s[1], 1, len[1]);  // 初始时模式串的哈希值
    for (i = 1; i <= len[0] - len[1] + 1; i++) {
        // 若两哈希值一致，则认为匹配
        if ((h0 - h1 * Exp[i - 1]) % M == 0)
            ans.push_back(i);
        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
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    ll i, j;

    // 初始化
    Exp[0] = 1;
    for (i = 1; i < N; i++)
        Exp[i] = Exp[i - 1] * b % M;

    cin >> (s[0] + 1) >> (s[1] + 1);
    len[0] = strlen(s[0] + 1), len[1] = strlen(s[1] + 1);

    match();

    return 0;
}
```

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 abcabcd 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：

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

```

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 $O(n^2)$。

```
void prefix_func1(char t[], int n) {
    int i, j;
    i = 2, j = 1;
    while (i <= n) {
        if (t[i] == t[j])  // 加1
            pi[i] = pi[i - 1] + 1, ++j;
        else {  // 开始减
            pi[i] = pi[i - 1];
            while (strncmp(t + i - pi[i] + 1, t + 1, pi[i]))
                --pi[i];
            j = pi[i] + 1;
        }
        ++i;
    }
}

```

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 $s[1\dots j]=s[i-j+1\dots i]$：

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

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 $s[1\ldots j]=s[i-j+1\ldots i]$ 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 $s[1\ldots k]=s[i-k+1\ldots i]$ 仍然满足。

$$\overbrace{\underbrace{s_1\ s_2}_{k}\ s_3\ s_4}^{j}\ \ldots\ \overbrace{s_{i-3}\ s_{i-2}\ \underbrace{s_{i-1}\ s_i}_{k}}^{j}\ s_{i+1}$$

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:

$$\overbrace{\underbrace{s_1\ s_2}_{k}\ \underbrace{s_3\ s_4}_{k}}^{j}$$

即 $k=\pi[j]$，而 $\pi[j]$ 之前已经求过了。

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

```

1.2.2 统计每个前缀出现次数

1.2.2.1 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。

```
// 统计s[]的每个前缀在s[]中出现的次数
inline void count_prefix(char s[], int n) {
    prefix_func(s, n);  // 计算前缀函数
    int i;
    for (i = 1; i <= n; i++)
        ++ans[pi[i]];
    // 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]]
    // 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
    for (i = n; i >= 1; i--)
        ans[pi[i]] += ans[i];
    // 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
    for (i = 1; i <= n; i++)
        ++ans[i];
}

```

1.2.2.2 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 $\pi[i]$。

```
// 统计t[]的每个前缀在s[]中出现的次数
inline void count_prefix(char s[], int n, char t[], int m) {
    // 连接字符串 t..#s..
    strcpy(link + 1, t + 1);
    link[m + 1] = '#';
```

```
    strcpy(link + m + 2, s + 1);
    // 计算前缀函数
    prefix_func(link, n + m + 1);

    int i;
    // 只关心#后面的pi值
    for (i = m + 2; i <= n + m + 1; i++)
        ++ans[pi[i]];
    // pi[i]的值不会超过t的长度，即i<=m
    for (i = m; i >= 1; i--)
        ans[pi[i]] += ans[i];
}
```

1.2.3 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。

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

1.2.4 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。

```
// 计算s的最短压缩表示，返回最短压缩的长度
int compress(char s[], int n) {
    prefix_func(s, n);
    int k = n - pi[n];
    if (n % k)
        return n;
    return k;
}
```

1.2.5 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。

```
// 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
void compute_automaton0(char t[], int n, int aut[][26]) {
    t[n + 1] = '#', t[n + 2] = 0, ++n;
    prefix_func(t, n);  // 先求t[]的前缀函数
    int i, j, c;
    // 0是初始状态，匹配长度为0；n是终止状态，完全匹配
    for (i = 0; i <= n; i++) {
```

```
        // 转移的因素--下一个字符
        for (c = 0; c < 26; c++) {
            j = i;
            // 通过不断跳转前缀来匹配下一个字符
            while (j >= 0 && c + 'a' != t[j + 1])
                j = pi[j];
            aut[i][c] = j + 1;
        }
    }
}

```

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 $O(|\Sigma|n)$。

```
// 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
void compute_automaton1(char t[], int n, int aut[][26]) {
    prefix_func(t, n);  // 先求t[]的前缀函数
    int i, c;
    // 0是初始状态，匹配长度为0；n是终止状态，完全匹配
    aut[0][t[1] - 'a'] = 1;
    for (i = 1; i <= n; i++) {
        // 转移的因素--下一个字符
        for (c = 0; c < 26; c++) {
            // 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
            if (c + 'a' == t[i + 1])
                aut[i][c] = i + 1;
            else
                aut[i][c] = aut[pi[i]][c];
        }
    }
}

```

1.2.6 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]。

```
// 自动机可能有问题

#include <bits/stdc++.h>

using namespace std;
const int N = 1e3 + 10;

int s[N];
int n, k;
int pi[N];
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
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
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

void prefix_func(int t[], int len) {
    int i, j;
    pi[0] = -1, pi[1] = 0;  // 确实没有找到任何相等的
    for (i = 1; i < len; ++i) {
        j = pi[i];
        while (j >= 0 && t[j + 1] != t[i + 1])  // 若不相等，找更小的新的j
            j = pi[j];
        pi[i + 1] = j + 1;  //最后得出pi[i+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
void compute_automaton1(int t[], int n, int aut[][N]) {
    t[n + 1] = -1;       // 结束符，结束符应是字母表中没有的符号
    prefix_func(t, n);  // 先求t[]的前缀函数
    int i, c;
    // 0是初始状态，匹配长度为0；n是终止状态，完全匹配
    aut[0][t[1]] = 1;
    for (i = 1; i <= n; i++) {
        // 转移的因素--下一个字符
        for (c = 1; c <= n; c++) {
            // 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
            if (c == t[i + 1])
                aut[i][c] = i + 1;
            else
                aut[i][c] = aut[pi[i]][c];
        }
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int i, j;
    cin >> n >> k;
    for (i = 1; i <= n; i++)
        cin >> s[i];

    compute_automaton1(s, n, aut);

    // 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
    for (i = 0; i <= n; i++)
        G[i][0] = i, K[i][0] = 0;
    // 类似于动态规划的递推
    for (j = 1; j <= k; j++) {
        for (i = 0; i <= n; i++) {
            int mid = aut[G[i][j - 1]][j];
            G[i][j] = G[mid][j - 1];
            K[i][j] = K[i][j - 1] + (n == mid) + K[mid][j - 1];
```

```
        }
    }

    cout << K[0][k] << endl;

    return 0;
}

```

1.2.7 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 $POPPOP$ 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（类似子问题）。之后再用上面的区间 dp，就是对的了。

```cpp
#include <bits/stdc++.h>

using namespace std;
const int N = 85;

char s[N];
int n;
int pi[N], pi2[N][N];  // pi2[i][j]表示以i为左端点，j处的前缀函数值
int f[N][N];           // f[i][j]表示s[i..j]被压缩后的最短长度

void prefix_func(char t[], int len) {
    int i, j;
    pi[0] = -1, pi[1] = 0;  // 确实没有找到任何相等的
    for (i = 1; i < len; ++i) {
        j = pi[i];
        while (j >= 0 && t[j + 1] != t[i + 1])  // 若不相等，找更小的新的j
            j = pi[j];
        pi[i + 1] = j + 1;  //最后得出pi[i+1]
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int i, j, k, l, r, m;

    while (cin >> (s + 1)) {
        if (s[1] == '*')
            break;
        n = strlen(s + 1);

        // 求pi2[i][j]
        for (i = 1; i <= n; i++) {
            prefix_func(s + i - 1, n - i + 1);
            for (j = i; j <= n; j++)
                pi2[i][j] = pi[j - i + 1];
        }

        // dp求f(l,r)，s[l..r]被压缩后的最短长度
```

```
        for (i = 1; i <= n; i++) {                // 枚举长度
            for (l = 1; l + i - 1 <= n; l++) {  // 枚举左端点
                r = l + i - 1;                  // 右端点
                f[l][r] = i;                    // 初始化为最坏情况
                int len = r - l + 1;            // 当前区间的长度
                // 如果本身完全是循环节
                k = len - pi2[l][r];
                if (len % k == 0)
                    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
                // 区间DP
                for (m = l; m < r; m++)
                    f[l][r] = min(f[l][r], f[l][m] + f[m + 1][r]);
            }
        }

        cout << f[1][n] << endl;
    }

    return 0;
}
```