南京工業大學
NANJING TECH UNIVERSITY

ICPC Template Manual

作者: 贺梦杰

October 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

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

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 pi[]) {
    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 pi[]) {
    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 $\pi[i+1]=\pi[i]+1$。

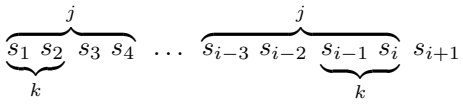
$$\underbrace{\overbrace{s_1\ s_2\ s_3}^{\pi[i]}\ \overbrace{s_4}^{s_4=s_{i+1}}}_{\pi[i+1]=\pi[i]+1}\ \dots\ \underbrace{\overbrace{s_{i-2}\ s_{i-1}\ s_i}^{\pi[i]}\ \overbrace{s_{i+1}}^{s_4=s_{i+1}}}_{\pi[i+1]=\pi[i]+1}$$

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 $s[1\dots j]=s[i-j+1\dots i]$：

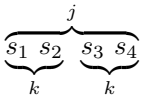
$$\overbrace{\underbrace{s_1\ s_2}_{j}\ s_3\ s_4}^{\pi[i]}\ \dots\ \overbrace{s_{i-3}\ s_{i-2}\ \underbrace{s_{i-1}\ s_i}_{j}}^{\pi[i]}\ s_{i+1}$$

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 $s[1\dots j]=s[i-j+1\dots i]$ 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。



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:



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
```

```
    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];
}

```

1.2.3.2 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 $\pi[i]$。

```
inline void count_prefix(char s[], int n, char t[], int m, int pi[]) {
    // 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, pi);

    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];
}

```

1.2.4 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。

```
// 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, pi);        // 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;
}

```

1.2.5 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 $k = i - \pi[i]$。

```
// 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, int pi[]) {
    prefix_func(s, n, pi);
    int k = n - pi[n];
    if (n % k)
        return n;
    return k;
}

```

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。

```
// 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 pi[], int aut[][26]) {
    prefix_func(t, n, pi);  // 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 pi[], int aut[][26]) {
    prefix_func(t, n, pi);  // 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];
        }
    }
}

```

1.2.7 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 $i \le n$ 且 $j \le k$ 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]。

```cpp
#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 n, int pi[]) {
    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]
    }
}

// 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 pi[], 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, pi);  // 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, pi, 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;
}

```

1.2.8 UVA11022 String Factoring

给定一个字符串 $s(|s| \le 80)$，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：

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

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。

```
#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 n, int pi[]) {
    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]
    }
}

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, pi2[i] + 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;
}

```

1.2.9 UVA12467 Secret word

给你一个串 $S(1 \le |S| \le 10^6)$，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。

1.3 AC 自动机

1.3.1 概述

AC 自动机是 **以 TRIE 的结构为基础**，结合 **KMP 的思想**建立的。

简单来说，建立一个 AC 自动机有两个步骤：
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。

然后就可以利用它进行多模式匹配了。

1.3.2 代码

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

using namespace std;
const int N = 1e6 + 10;

struct AC {
    int tr[N][26], cnt;  // 字典树/自动机; 节点数
    int exist[N];        // 标记以该节点为结尾的字符串有几个(因为题中说了输入可能有相同的模式串)
    // 构建最初的字典树
    void insert(char s[], int l) {
        int p = 0, i;  // p表示当前所处节点，p=0时为起点(空串)
        for (i = 0; i < l; i++) {
            int c = s[i] - 'a';
            if (!tr[p][c])
                tr[p][c] = ++cnt;  // 如果没有就添加节点
            p = tr[p][c];          // 每一步结束p都会指向最后的已存在的节点
        }
        exist[p]++;  // 记录以该节点为结尾的字符串的个数
    }

    int fail[N];   // fail指针
    queue<int> q;  // 用于bfs
    // 构建自动机及fail指针，此时tr就进化为一个自动机了
    void build() {
        int i;
        // 队列中一开始是根节点的儿子节点们，儿子们的fail指向根节点(0)
        for (i = 0; i < 26; i++)
            if (tr[0][i])
                q.push(tr[0][i]);
        // 开始bfs
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (i = 0; i < 26; i++) {
                // 与kmp自动机中的dp思想很像
                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];
            }
        }
    }

    bool vis[N];  // 自动机是否已经匹配过
    int query(char s[], int l) {
        int u = 0, res = 0, i, j;
        for (i = 0; i < l; i++) {
            u = tr[u][s[i] - 'a'];  // 转移到下一个状态
            // 检查每一个后缀是否可以匹配，且对于每个模式串只匹配一次
            for (j = u; !vis[j]; j = fail[j])
                res += exist[j], vis[j] = 1;
```

```
        }
        return res;
    }
} ac;

int n;
char s[N];

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

    int i;
    cin >> n;
    for (i = 1; i <= n; i++) {
        cin >> s;
        ac.insert(s, strlen(s));
    }
    ac.build();

    cin >> s;
    cout << ac.query(s, strlen(s)) << endl;

    return 0;
}
```

1.3.3 洛谷 P5357 【模板】AC 自动机（二次加强版）

给你一个文本串 S 和 n 个模式串 $T_{1..n}$，请你分别求出每个模式串 T_i 在 S 中出现的次数。

解法：AC 自动机，树形 DP

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

using namespace std;
const int N = 2e5 + 10, M = 2e6 + 10;

int n;
int mp[N];
string s[N];       // 模式串
vector<int> g[N];  // 图
int ans[N];
bool vis[N];
char t[M];

struct AC {
    int tr[N][26], cnt;  // 字典树/自动机; 节点数
    bool exist[N];       // 标记以该节点为结尾的字符串是否存在
    // 构建最初的字典树
    int insert(string s, int l) {
        int p = 0, i;  // p表示当前所处节点，p=0时为起点(空串)
        for (i = 0; i < l; i++) {
            int c = s[i] - 'a';
            if (!tr[p][c])
                tr[p][c] = ++cnt;  // 如果没有就添加节点
            p = tr[p][c];          // 每一步结束p都会指向最后的已存在的节点
        }
        exist[p] = 1;  // 标记以该节点为结尾的字符串存在
        return p;
    }

    int fail[N];   // fail指针
    queue<int> q;  // 用于bfs
    // 构建自动机及fail指针，此时tr就进化为一个自动机了
    void build() {
        int i;
```

```
35          // 队列中一开始是根节点的儿子节点们，儿子们的fail指向根节点(0)
36          for (i = 0; i < 26; i++)
37              if (tr[0][i])
38                  q.push(tr[0][i]);
39          // 开始bfs
40          while (!q.empty()) {
41              int u = q.front();
42              q.pop();
43              g[fail[u]].push_back(u);  // 按fail指针反向构建树
44              for (i = 0; i < 26; i++) {
45                  // 与kmp自动机中的dp思想很像
46                  if (tr[u][i])
47                      fail[tr[u][i]] = tr[fail[u]][i], q.push(tr[u][i]);
48                  else
49                      tr[u][i] = tr[fail[u]][i];
50              }
51          }
52      }
53
54      void query(char s[], int l) {
55          int u = 0, i, j;
56          for (i = 0; i < l; i++) {
57              u = tr[u][s[i] - 'a'];  // 转移到下一个状态（最大匹配后缀，所以下面还要跳转至更短的后缀）
58              ++ans[u];
59          }
60      }
61  } ac;
62
63  // 树形DP思想，将长串出现的次数累加至其fail指针能跳到的短串
64  void dfs(int u) {
65      if (vis[u])
66          return;
67      vis[u] = 1;
68      for (auto v : g[u]) {
69          dfs(v);
70          ans[u] += ans[v];
71      }
72  }
73
74  int main() {
75      ios::sync_with_stdio(0);
76      cin.tie(0);
77
78      int i;
79      cin >> n;
80      for (i = 1; i <= n; i++) {
81          cin >> s[i];
82          mp[i] = ac.insert(s[i], s[i].length());
83      }
84      ac.build();
85      cin >> t;
86      ac.query(t, strlen(t));
87      for (i = 0; i <= ac.cnt; i++)
88          dfs(i);
89      for (i = 1; i <= n; i++)
90          cout << ans[mp[i]] << endl;
91
92      return 0;
93  }
```

1.3.4 子矩阵匹配

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

首先对小矩阵 t[][] 构建一个 AC 自动机，然后将大矩阵的每一行丢到 AC 自动机里跑，并求出每一个 s[i][j] 处

匹配到自动机的哪个节点记录到 pos[i][j]。接下来只要将 s[][] 的每一个位置当作起点，竖着向下匹配即可。

```
    /*
子矩阵匹配
AC自动机
*/

#include <bits/stdc++.h>

using namespace std;
const int N = 1e3 + 10, M = 1e2 + 10;

char s[N][N], t[M][M];
int n, m, x, y;
int pos[N][N];

// 对AC自动机全面初始化
struct AC {
    void init() {
        int i;
        for (i = 0; i <= cnt; i++) {
            memset(tr[i], 0, sizeof(tr[i]));
            memset(exist[i], 0, sizeof(exist[i]));
            fail[i] = 0;
        }
        while (!q.empty())
            q.pop();
        cnt = 0;
    }
    int tr[M * M][26], cnt;  // 字典树/自动机; 节点数
    bool exist[M * M][200];
    // 构建最初的字典树
    void insert(char s[], int l, int id) {
        int p = 0, i;  // p表示当前所处节点，p=0时为起点(空串)
        for (i = 0; i < l; i++) {
            int c = s[i] - 'a';
            if (!tr[p][c])
                tr[p][c] = ++cnt;  // 如果没有就添加节点
            p = tr[p][c];          // 每一步结束p都会指向最后的已存在的节点
        }
        exist[p][id] = 1;
    }

    int fail[M * M];  // fail指针
    queue<int> q;     // 用于bfs
    // 构建自动机及fail指针，此时tr就进化为一个自动机了
    void build() {
        int i;
        // 队列中一开始是根节点的儿子节点们，儿子们的fail指向根节点(0)
        for (i = 0; i < 26; i++)
            if (tr[0][i])
                q.push(tr[0][i]);
        // 开始bfs
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (i = 0; i < 26; i++) {
                // 与kmp自动机中的dp思想很像
                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];
            }
        }
    }

    void query(char s[], int l, int pos[]) {
```

```
66          int u = 0, i, j;
67          for (i = 0; i < l; i++) {
68              u = tr[u][s[i] - 'a'];  // 转移到下一个状态（最大匹配后缀，所以下面还要跳转至更短的后缀）
69              pos[i] = u;
70          }
71      }
72  } ac;
73
74  int solve() {
75      int i, j, k, ans = 0;
76      for (i = 1; i <= n; i++) {
77          for (j = 0; j < m; j++) {
78              bool ok = 1;
79              for (k = 0; k < x; k++) {
80                  if (!ac.exist[pos[i + k][j]][k + 1]) {
81                      ok = 0;
82                      break;
83                  }
84              }
85              ans += ok;
86          }
87      }
88      return ans;
89  }
90
91  int main() {
92      ios::sync_with_stdio(0);
93      cin.tie(0);
94
95      int T, i, j, k;
96      scanf("%d", &T);
97      while (T--) {
98          scanf("%d%d", &n, &m);
99          for (i = 1; i <= n; i++)
100             scanf("%s", s[i]);
101         scanf("%d%d", &x, &y);
102         for (i = 1; i <= x; i++)
103             scanf("%s", t[i]);
104
105         // 构建自动机
106         for (i = 1; i <= x; i++)
107             ac.insert(t[i], y, i);
108         ac.build();
109         for (i = 1; i <= n; i++)
110             ac.query(s[i], m, pos[i]);
111
112         // 求
113         int ans = solve();
114         printf("%d\n", ans);
115
116         // 初始化
117         ac.init();
118     }
119
120     return 0;
121 }
```

1.4 后缀数组

1.4.1 概述

后缀数组（Suffix Array）主要是两个数组：sa 和 rk 。
其中，$sa[i]$ 表示将所有后缀排序后第 i 小的后缀的编号。$rk[i]$ 表示后缀 i 的排名。
这两个数组满足性质：$sa[rk[i]] = rk[sa[i]] = i$ 。

1.4.2 后缀数组求法

用倍增的思想，再用基数排序优化。

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

using namespace std;
const int N = 1e6 + 10;  // 拼接开双倍

int n;
char s[N];
int sa[N], rk[N];
int cnt[N];  // 计数=>前缀和

struct Tri {
    int key[2], id;
    bool operator<(const Tri& rhs) const {
        return key[0] != rhs.key[0] ? key[0] < rhs.key[0] : key[1] < rhs.key[1];
    }
} tri[N], b[N];

void getSA() {
    int i, w, k, mx;
    for (i = 1; i <= n; i++)
        rk[i] = s[i];  // 初始化大小
    for (w = 1; w < n; w <<= 1) {
        for (i = 1; i <= n; ++i)
            tri[i] = {rk[i], i + w <= n ? rk[i + w] : 0, i};  // 倍增rk
        for (k = 1; k >= 0; --k) {                            // 基数排序
            memset(cnt, 0, sizeof(cnt)), mx = 0;
            for (i = 1; i <= n; ++i)
                ++cnt[tri[i].key[k]], mx = max(mx, tri[i].key[k]);
            for (i = 1; i <= mx; ++i)
                cnt[i] += cnt[i - 1];
            for (i = n; i >= 1; --i)
                b[cnt[tri[i].key[k]]--] = tri[i];
            memcpy(tri, b, sizeof(tri));
        }
        for (rk[tri[1].id] = k = 1, i = 2; i <= n; ++i) {
            if (tri[i - 1] < tri[i])  // 去重
                ++k;
            rk[tri[i].id] = k;  // 得出当前的rk
        }
    }
    for (i = 1; i <= n; ++i)  // 由rk算后缀数组sa
        sa[rk[i]] = i;
}

int main() {
    int i;
    scanf("%s", s + 1);
    n = strlen(s + 1);
    s[n + 1] = 0;  // 串的终结符，不同情况下值不一样，要求输入中没有且比输入值都小
    getSA();
    for (i = 1; i <= n; i++)
        printf("%d ", rk[i]);
    printf("\n");
    for (i = 1; i <= n; i++)
```

```
55          printf("%d ", sa[i]);
56      printf("\n");
57
58      return 0;
59  }
60
61  /*
62  aabaaaab
63  4 6 8 1 2 3 5 7
64  4 5 6 1 7 2 8 3
65  */
```

1.4.3 循环串的排序问题

1.4.3.1 题目描述

给你一个串 S，S 绕成一圈，然后 S 就有了很多种读法。现在要求你将所有读法的字符串排序，读出最后一列字符。
例如‘JSOI07’，可以读作：JSOI07 SOI07J OI07JS I07JSO 07JSOI 7JSOI0 把它们按照字符串的大小排序：07JSOI 7JSOI0 I07JSO JSOI07 OI07JS SOI07J 读出最后一列字符：I0O7SJ

1.4.3.2 解法

将字符串 S 复制一份变成 SS，对 SS 求后缀数组，则得到所有读法的排名。但是要注意，若 $sa[i] > n$(n 是 S 的长度)，则该读法是无效的，应直接跳过。
拼接后的字符串 SS 的每一个后缀长度并不一定刚好等于读出的长度 n，为什么可以直接用后缀数组而无需修改呢？对于有效的读法（长度大于等于 n），考虑两种情况：1. 在比较长度不到 n 的时候已经发生不等，此时顺序显然是正确的；2. 前 n 个字符都相等，后面发生了不等，其实后面的大小关系不影响结果，因为前 n 个已经相等了，说明这两个读法是相等的，而他们的相对位置怎样都是可以的。

```
1       #include <bits/stdc++.h>
2
3   using namespace std;
4   const int N = 2e5 + 10;  // 拼接开双倍
5
6   int n;
7   char s[N];
8   int sa[N], rk[N];
9   int cnt[N];  // 计数=>前缀和
10
11  // getSA()
12
13  int main() {
14      ios::sync_with_stdio(0);
15      cin.tie(0);
16
17      int i;
18      cin >> s + 1;
19      n = strlen(s + 1);
20      strncpy(s + 1 + n, s + 1, n);
21      n *= 2;
22      getSA();
23      n /= 2;
24      for (i = 1; i <= 2 * n; ++i)
25          if (sa[i] <= n)
26              cout << s[sa[i] + n - 1];
27      cout << endl;
28
29      return 0;
30  }
```

1.4.4 在字符串中找子串

任务是在线地在主串 S 中寻找模式串 T 。在线的意思是，我们已经预先知道知道主串 S ，但是当且仅当询问时才知道模式串 T 。我们可以先构造出 S 的后缀数组，然后查找子串 T 。若子串 T 在 S 中出现，它必定是 S 的

一些后缀的前缀。因为我们已经将所有后缀排序了，我们可以通过在 sa 数组中二分 T 来实现。比较子串 T 和当前后缀的时间复杂度为 $O(|T|)$ ，因此找子串的时间复杂度为 $O(|T|\log|S|)$ 。注意，如果该子串在 S 中出现了多次，每次出现都是在 sa 数组中相邻的。因此出现次数可以通过再次二分找到，输出每次出现的位置也很轻松。

因为强制在线，所以不能用 AC 自动机
那这与 KMP 由什么区别呢?
这里用 KMP 每次都要对模式串求 pi，而后缀数组只要对主串求一次
KMP 复杂度：O(M+N)；后缀数组复杂度：O(MlogN)。
显然，当 m 很小而 n 很大的时候后缀数组的优势就体现出来了

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

using namespace std;
const int N = 1e2 + 10;  // 拼接开双倍

int n, m;          // s和t的长度
char s[N], t[N];  // 在主串s中在线的寻找t
int sa[N], rk[N];
int cnt[N];  // 计数=>前缀和

// getSA()

// 若s的后缀p大于t返回正数，小于返回负数，相等返回0
int cmp(int p) {
    int i;
    for (i = 1; i <= m && p <= n; ++p, ++i)
        if (s[p] != t[i])
            return s[p] - t[i];
    return i == m + 1 ? 0 : -1;
}

// 当前区间[l,r]，找等于模式串t的后缀的最大排名
int find1(int l, int r) {
    if (l > r)
        return 0;
    int mid = (l + r) / 2;
    int x = cmp(sa[mid]);
    if (x > 0)  // 后缀大于t
        return find1(l, mid - 1);
    else if (x < 0)
        return find1(mid + 1, r);
    else
        return max(mid, find1(mid + 1, r));
}

// 找小于模式串t的后缀的最大排名
int find2(int l, int r) {
    if (l > r)
        return 0;
    int mid = (l + r) / 2;
    int x = cmp(sa[mid]);
    if (x >= 0)  // 后缀大于等于t
        return find2(l, mid - 1);
    else  // 后缀小于t
        return max(mid, find2(mid + 1, r));
}

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

    int q;  // q次询问
    cin >> q >> (s + 1);
    n = strlen(s + 1);
    getSA();
    while (q--) {
```

```
57          cin >> (t + 1);
58          m = strlen(t + 1);
59          int r1 = find1(1, n);
60          int r2 = find2(1, n);
61          int ans = r1 - r2;
62          if (ans > 0)
63              cout << ans << endl;
64          else
65              cout << 0 << endl;
66      }
67
68      return 0;
69  }
```

1.4.5 从字符串首尾取字符最小化字典序

题意：给你一个字符串，每次从首或尾取一个字符放到新字符串末尾，取完为止，问所有能够组成的字符串中字典序最小的一个。

解法一：贪心 + 暴力，每次从两边向中间取字符，直到两个串不相等，则把小的那个全部加到新串末尾，如果到最后相遇都相等，则直接全取

解法二：由于需要在原串后缀与反串后缀构成的集合内比较大小，可以将反串拼接在原串后，并在中间加上一个没出现过的字符（如 #，代码中可以直接使用空字符），求后缀数组，即可 O(1) 完成这一判断。

```
1   #include <bits/stdc++.h>
2
3   using namespace std;
4   const int N = 1e6 + 10;  // 拼接开双倍
5
6   int n;
7   char s[N], ans[N];
8   int sa[N], rk[N];
9   int cnt[N];  // 计数=>前缀和
10
11  // getSA()
12
13  int main() {
14      int i, j, m, k;
15      scanf("%d", &n);
16      for (i = 1; i <= n; ++i) {
17          char ch = getchar();
18          while (!isalpha(ch))
19              ch = getchar();
20          s[i] = ch;
21      }
22      s[n + 1] = '#';
23      for (i = 1; i <= n; ++i)
24          s[n + 1 + i] = s[n + 1 - i];
25      m = n, n = 2 * n + 1;
26      getSA();
27
28      i = 1, j = m + 2, k = 1;
29      while (k <= m) {
30          if (rk[i] < rk[j])
31              ans[k++] = s[i++];
32          else
33              ans[k++] = s[j++];
34      }
35
36      for (i = 1; i <= m; ++i) {
37          printf("%c", ans[i]);
38          if (i % 80 == 0)
39              printf("\n");
```

```
40      }
41      if ((i - 1) % 80)
42          printf("\n");
43
44      return 0;
45  }
```

1.4.6 height 数组

1.4.6.1 LCP（最长公共前缀）

两个字符串 S 和 T 的 LCP 就是最大的 x（$x \leq \min(|S|, |T|)$）使得 $S_i = T_i$ $(\forall\ 1 \leq i \leq x)$。
下文中以 $lcp(i, j)$ 表示后缀 i 和后缀 j 的最长公共前缀（的长度）。

1.4.6.2 height 数组的定义

$height[i] = lcp(sa[i], sa[i-1])$，即第 i 名的后缀与它前一名的后缀的最长公共前缀。
$height[1]$ 可以视作 0。

1.4.6.3 O(n) 求 height 数组需要的一个引理

这里先不给出这个引理是什么，主要是为了防止证明（推导）过程中混乱。
首先要是搞清楚 $height[i]$ 中的 i 是**排名**。我们令后缀 $i-1$ 为 aAD，其首字母为 a，$|A| = height[rk[i-1]] - 1$，$D$ 为剩余部分。这里 $height[rk[i-1]]-1$ 看起来非常绕，其实 $height[rk[i-1]]-1 == lcp(sa[rk[i-1]], sa[rk[i-1]-1]) - 1$，这就变得清晰多了，$sa[rk[i-1]]$ 就表示后缀 $i-1$，$sa[rk[i-1]-1]$ 表示排名比后缀 $i-1$ 小 1 的后缀。那么 aA 其实就是后缀 $i-1$ 和排名比后缀 $i-1$ 小 1 的后缀的 LCP，而 A 就是 LCP 拿掉首字母 a。
由于 $sa[rk[i-1]-1]$ 与 $sa[rk[i-1]]$ 有一段 LCP 为 aA，所以我们可以令 $sa[rk[i-1]-1]$ 为 aAB，则 $sa[rk[i-1]-1]+1$ 为 AB，且 $rank(AD) \geq rank(AB)+1$（由 $rank$ 关系）。而由后缀 $i-1$ 为 aAD 我们还可以知道后缀 i 为 AD，我们令比后缀 i 排名小 1 的后缀为 X 即 $X = sa[rk[i]-1]$，则我们有 $rank(AD) == rank(X)+1$。由 $rank(AD) \geq rank(AB)+1$ 和 $rank(AD) == rank(X)+1$，我们可以得到 $rank(AD) > rank(X) >= rank(AB)$，所以我们可以确定 X 一定含有前缀 A，那我们就可以令 $X = AC$，则 $lcp(sa[rk[i]], sa[rk[i]-1]) \geq |A|$。
最后，我们得到了这个**引理：** $height[rk[i]] \geq height[rk[i-1]] - 1$。

1.4.6.4 实现

```
1   // O(n)求height数组
2   void getHeight() {
3       int i, k;
4       // 预处理height
5       for (i = 1, k = 0; i <= n; ++i) {
6           k -= k > 0;                                  // ht[rk[i]] >= ht[rk[i]-1] - 1 且 ht[] >= 0
7           while (s[i + k] == s[sa[rk[i] - 1] + k])  // 暴力求lcp
8               ++k;
9           ht[rk[i]] = k;
10      }
11  }
```

1.4.7 height 数组求两子串最长公共前缀

$lcp(sa[i], sa[j]) = \min\{height[i+1..j]\}$
感性理解：如果 $height$ 一直大于某个数，前这么多位就一直没变过；反之，由于后缀已经排好序了，不可能变了之后变回来。
严格证明可以参考 [2004 后缀数组 by. 徐智磊][1]。
有了这个定理，求两子串最长公共前缀就转化为了 RMQ 问题。

如果我们要求后缀 x 和后缀 y 的 lcp，设 $rk[x] < rk[y]$，则 $lcp(x, y) = \min\{height[rk[x]+1 \ldots rk[y]]\}$

```
1   #include <bits/stdc++.h>
2
3   using namespace std;
4   const int N = 1e2 + 10;  // 拼接开双倍
5
```

```
6  int n;
7  char s[N];
8  int sa[N], rk[N];
9  int cnt[N];      // 计数=>前缀和
10 int ht[N];       // height数组，ht[排名]
11 int rmq[N][20];  // RMQ，rmq[i][j]表示以i为起点长度为2^j的区间里的最大/小值。
12 int Log2[N];     // 预处理的log2[]取整
13
14 // getSA()
15
16 void init() {
17     // 预处理Log2[]
18     Log2[1] = 0;
19     for (int i = 2; i < N; ++i)
20         Log2[i] = Log2[i / 2] + 1;
21 }
22
23 // O(n)求height数组
24 void getHeight() {
25     int i, j, k;
26     // 预处理height
27     for (i = 1, k = 0; i <= n; ++i) {
28         k -= k > 0;                                  // ht[rk[i]] >= ht[rk[i]-1] - 1 且 ht[] >= 0
29         while (s[i + k] == s[sa[rk[i] - 1] + k])  // 暴力求lcp
30             ++k;
31         ht[rk[i]] = k;
32     }
33
34     // 预处理rmq
35     for (i = 1; i <= n; ++i)
36         rmq[i][0] = ht[i];
37     k = Log2[n];
38     for (j = 1; j <= k; ++j)
39         for (i = 1; i <= n; ++i)
40             rmq[i][j] = min(rmq[i][j - 1], rmq[i + (1 << (j - 1))][j - 1]);
41 }
42
43 // 求后缀x和后缀y的lcp。RMQ问题
44 int lcp(int x, int y) {
45     if (x == y)  // 自身和自身的lcp等于自身
46         return n - x + 1;
47     if (rk[x] > rk[y])
48         swap(x, y);
49     x = rk[x] + 1, y = rk[y];  // 注意是height[ rk[x]+1 .. rk[y] ]
50     int k = Log2[y - x + 1];
51     return min(rmq[x][k], rmq[y - (1 << k) + 1][k]);
52 }
53
54 int main() {
55     init();
56     int i, x, y;
57     scanf("%s", s + 1);
58     n = strlen(s + 1);
59     getSA(), getHeight();
60     while (1) {
61         scanf("%d%d", &x, &y);
62         int len = lcp(x, y);
63         for (i = 0; i < len; ++i)
64             printf("%c", s[x + i]);
65         printf("\n");
66     }
67
68     return 0;
69 }
```

1.4.8 height 数组比较子串

假设需要比较的是 $A = S[a..b]$ 和 $B = S[c..d]$ 的大小关系。若 $lcp(a, c) \geq \min(|A|, |B|)$ ，$|A| < |B| \iff A < B$ (谁长谁大，等长相等)。否则，$rk[a] < rk[c] \iff A < B$ 。

```
// 比较子串s[a..b]和s[c..d]，大于返回正数，等于返回0，小于返回负数
int subCmp(int a, int b, int c, int d) {
    int A = b - a + 1, B = d - c + 1;
    if (lcp(a, c) >= min(A, B))  //lcp长度大于等于较短串
        return A - B;
    return rk[a] - rk[c];  // 否则直接比较两后缀的rk
}
```

1.4.9 height 数组求本质不同的子串个数

1.4.9.1 本质不同的子串个数

子串就是后缀的前缀，所以可以枚举每个后缀，计算前缀总数，再减掉重复。
“前缀总数” 其实就是子串个数，为 $n(n+1)/2$ 。
如果按后缀排序的顺序枚举后缀，每次新增的子串就是除了与上一个后缀的 LCP 剩下的前缀。这些前缀一定是新增的，否则会破坏 $lcp(sa[i], sa[j]) = \min\{height[i+1..j]\}$ 的性质。只有这些前缀是新增的，因为 LCP 部分在枚举上一个前缀时计算过了。
所以答案为：$\frac{n(n+1)}{2} - \sum_{i=2}^{n} height[i]$

```
// O(n)求本质不同的子串个数
int diff() {
    int sum = n * (n + 1) / 2, i;
    for (i = 1; i <= n; ++i)
        sum -= ht[i];
    return sum;
}
```

1.4.9.2 每个出现次数的本质不同的子串个数

令 $app[i]$ 表示恰好出现 i 次的不同子串个数。$last[i]$ 是一个滚动数组，表示比当前后缀排名小 1 的后缀的每个前缀已经出现的次数。本质思想和上面一样，上面是把所有 $height$ 合起来了，这里需要每一个后缀分开来求

```
// O(n^2)求每个出现次数的本质不同的子串个数
void diff() {
    int i, j;
    for (i = 1; i <= n; ++i) {
        int len = n - sa[i] + 1;  // 当前前缀长度
        for (j = 1; j <= ht[i]; ++j) {
            --app[last[j]];  // 排名更小的后缀中对应的前缀在当前后缀中再次出现了，所以更新app[]，并且为下一个排名更新last
            ++last[j];
            ++app[last[j]];
        }
        for (; j <= len; ++j)  // 初次出现的子串
            last[j] = 1, ++app[1];
    }
}
```

1.4.10 height 数组求至少出现 k 次的最长子串 (可重叠)

求一个串中至少出现 k 次的最长子串。

出现至少 k 次意味着后缀排序后有至少连续 k 个后缀的 LCP 是这个子串。
所以，求出每相邻 $k-1$ 个 $height$ 的最小值，再求这些最小值的最大值就是答案。
可以使用单调队列 $O(n)$ 解决，但使用其它方式也足以 AC。

```
/*
题：洛谷 P2852 [USACO06DEC]牛奶模式Milk Patterns
注意虽然元素个数n<=2e4，但是其值域为[0,1e6]。
值域包含0，那么0就不能作为结束符，故设终结符为-1
```

```
*/
deque<pair<int, int>> q;  // 单调队列
int solve() {
    int ans = 0, i;
    for (i = 1; i <= n; ++i) {
        while (!q.empty() && q.back().second > ht[i])
            q.pop_back();
        q.push_back({i, ht[i]});
        if (q.front().first <= i - k + 1)
            q.pop_front();
        if (i >= k)
            ans = max(ans, q.front().second);
    }
    return ans;
}
```

若改为恰好出现 k 次，基本做法不变。令 $mn = \min\{ht[i+1 \ldots i+k-1]\}$，只要 $ht[i] \le mn$ 且 $ht[i+k] \le mn$，则当前的 mn 就是符合条件的。

1.4.11 height 数组求不重叠的最长重复子串

先二分答案，把题目变成判定性问题：判断是否存在两个长度为 len 的子串是相同的，且不重叠。解决这个问题的关键还是利用 height 数组。把排序后的后缀分成若干组，其中每组的后缀之间的 height 值都不小于 len。（表示当前组中存在长度为 len 的重复子串）
然后对于每组后缀，只须判断每个后缀的 sa 值的最大值和最小值之差是否不小于 len。（差值不小于 len 也就不会重叠）

```
// 返回不重叠的最长重复出现的子串的长度
int solve() {
    int mx, mn, i, l = 1, r = n, mid, ans = 0, ok;  // mx和mn是每组后缀最大和最小的后缀起点
    while (l <= r) {
        ok = mx = mn = 0;
        mid = (l + r) / 2;                                  // 当前长度
        for (i = 1, ht[n + 1] = 0; i <= n + 1; ++i) {  // 为了处理最后一组，这里做到了n+1。所以要注意初始化问题
            if (ht[i] < mid) {                              // 已经到了下一个分组，结算上一分组
                if (mx - mn >= mid)                         // 不重叠
                    ans = max(ans, mid), ok = 1;
                mx = mn = sa[i];
            }
            mx = max(mx, sa[i]), mn = min(mn, sa[i]);
        }
        if (ok)  // 若存在这一长度的符合条件的子串
            l = mid + 1;
        else
            r = mid - 1;
    }
    return ans;
}
```

1.4.12 height 数组求最长回文子串

穷举每一位，然后计算以这个字符为中心的最长回文子串。注意这里要分两种情况，一是回文子串的长度为奇数，二是长度为偶数。两种情况都可以转化为求一个后缀和一个反过来写的后缀的最长公共前缀。具体的做法是：将整个字符串反过来写在原字符串后面，中间用一个特殊的字符隔开。这样就把问题变为了求这个新的字符串的某两个后缀的最长公共前缀。

```
// 返回最长回文串的长度
int palindr() {
    s[n + 1] = 1;                                  // 介于正串和反串之间的分隔符
    reverse_copy(s + 1, s + 1 + n, s + n + 2);  // 拼接反串
    n = n * 2 + 1;
    getSA(), getHeight();
    n = (n - 1) / 2;
    int i, ans = 1;
    for (i = 1; i <= n; ++i) {
```

```
        if (1 < i && i < n) {  //奇回文串
            int len = lcp(i + 1, 2 * n + 1 - i + 2) * 2 + 1;
            ans = max(ans, len);
        }
        if (i < n) {  // 偶回文串
            int len = lcp(i + 1, 2 * n + 1 - i + 1) * 2;
            ans = max(ans, len);
        }
    }
    return ans;
}
```

1.4.13 height 数组求循环节

做法比较简单, 穷举字符串 S 的长度 k, 然后判断是否满足。判断的时候, 先看字符串 L 的长度能否被 k 整除, 再看 suffix(1) 和 suffix(k+1) 的最长公共前缀是否等于 n-k。在询问最长公共前缀的时候,suffix(1) 是固定的, 所以 RMQ 问题没有必要做所有的预处理, 只需求出 height 数组中的每一个数到 height[rank[1]] 之间的最小值即可。整个做法的时间复杂度为 O(n)。显然, **前缀函数**更简单高效

1.4.14 height 数组求重复次数最多的循环子串

先穷举长度 L, 然后求长度为 L 的子串最多能连续出现几次。首先连续出现 1 次是肯定可以的, 所以这里只考虑至少 2 次的情况。假设在原字符串中连续出现 2 次, 记这个子字符串为 S, 那么 S 肯定包括了字符 r[0], r[L], r[L*2], r[L*3], ... 中的某相邻的两个。所以只须看字符 r[L*i] 和 r[L*(i+1)] 往前和往后各能匹配到多远, 记这个总长度为 K, 那么这里连续出现了 K/L+1 次。最后看最大值是多少。

```
// 返回最多出现的次数
int solve() {
    int ans = 1, len, i;  // 最少出现一次
    s[n + 1] = 1;
    reverse_copy(s + 1, s + 1 + n, s + 2 + n);
    n = n * 2 + 1;
    getSA(), getHeight();
    n /= 2;
    for (len = 1; len <= n; ++len) {
                                    // 穷举长度
        for (i = 0; 1 + (i + 1) * len <= n; ++i) {
                                    // 穷举包含点
            int tot = lcp(1 + i * len, 1 + (i + 1) * len) + lcp(2 * n + 2 - (1 + i * len), 2 * n
    + 2 - (1 + (i + 1) * len)) - 1;  // 向前向后总共匹配了多少
            ans = max(ans, tot / len + 1);
        }
    }
    return ans;
}
```

1.4.15 height 数组求最长公共子串

给定两个字符串 A 和 B, 求最长公共子串。
算法分析:
字符串的任何一个子串都是这个字符串的某个后缀的前缀。求 A 和 B 的最长公共子串等价于求 A 的后缀和 B 的后缀的最长公共前缀的最大值。如果枚举 A 和 B 的所有的后缀, 那么这样做显然效率低下。由于要计算 A 的后缀和 B 的后缀的最长公共前缀, 所以先将第二个字符串写在第一个字符串后面, 中间用一个没有出现过的字符隔开, 再求这个新的字符串的后缀数组。观察一下, 看看能不能从这个新的字符串的后缀数组中找到一些规律。以 A= “aaaba” ,B= “abaa” 为例, 如图 8 所示。那么是不是所有的 height 值中的最大值就是答案呢? 不一定! 有可能这两个后缀是在同一个字符串中的, 所以**实际上只有当 suffix(sa[i-1]) 和 suffix(sa[i]) 不是同一个字符串中的两个后缀时,height[i] 才是满足条件的。而这其中的最大值就是答案**。记字符串 A 和字符串 B 的长度分别为 |A| 和 |B|。求新的字符串的后缀数组和 height 数组的时间是 O(|A|+|B|), 然后求排名相邻但原来不在同一个字符串中的两个后缀的 height 值的最大值, 时间也是 O(|A|+|B|), 所以整个做法的时间复杂度为 O(|A|+|B|)。时间复杂度已经取到下限, 由此看出, 这是一个非常优秀的算法。

```
/*
t1和t2是两个输入串，它们的长度是len1和len2，将它们拼接成新字符串s(中间加一个分隔符)
*/
```

```
// 返回最长公共子串的长度
int lcs() {
    strcpy(s + 1, t1 + 1);
    s[len1 + 1] = '$';  // 分隔符
    strcpy(s + len1 + 2, t2 + 1);
    n = len1 + 1 + len2;  // 拼接串的长度
    getSA(), getHeight();
    int i, ans = 0, p;  // ans是长度，p是起始位置
    for (i = 2; i <= n; ++i) {
        int x = sa[i], y = sa[i - 1];
        if (x > y)
            swap(x, y);
        if (x <= len1 && y > len1)  // 分布在分隔符两侧，即属于不同串
            if (ans < ht[i])
                ans = ht[i], p = x;
    }

    return ans;
}
```

1.4.16 小常数模板

相比上面常数更小。
关键注意几点：
1. 首先将字符数组 s[] **转换**为 int 型的 r[]
2. da() 中 r[] 是从 0 到 n-1，且要求 r[n-1] 为 0，所有 r[i] 小于形参 m。所以，我们传入的**形参要做一些变化**。
3. 在这之后，所有的一切（下标、值域等）都和原模板一模一样。

```
#include <bits/stdc++.h>
using namespace std;

const int N = 1e6 + 10;
char s[N];
int r[N], n;                // 输入数组及其长度
int sa[N], rk[N], ht[N];  // 后缀数组，rank数组，height数组
// wa/wb是为da()开辟的；wv[i]相当于第一关键字的rk[i]；bs是基数排序的前缀和。（内部）
int wa[N], wb[N], wv[N], bs[N];
int cmp(int* r, int a, int b, int l) {  // 去重（内部）
    return r[a] == r[b] && r[a + l] == r[b + l];
}
// r[0..n-1]是待排序数组；sa[1..n]后缀数组；n是r[]的长度；m是r[]的最大值+1
void da(int* r, int* sa, int n, int m) {
    int i, j, p, *x = wa, *y = wb, *t;
    for (i = 0; i < m; i++)
        bs[i] = 0;
    for (i = 0; i < n; i++)
        bs[x[i] = r[i]]++;
    for (i = 1; i < m; i++)
        bs[i] += bs[i - 1];
    for (i = n - 1; i >= 0; i--)
        sa[--bs[x[i]]] = i;
    for (j = 1, p = 1; p < n; j *= 2, m = p) {
        for (p = 0, i = n - j; i < n; i++)
            y[p++] = i;
        for (i = 0; i < n; i++)
            if (sa[i] >= j)
                y[p++] = sa[i] - j;
        for (i = 0; i < n; i++)
            wv[i] = x[y[i]];
        for (i = 0; i < m; i++)
            bs[i] = 0;
        for (i = 0; i < n; i++)
            bs[wv[i]]++;
        for (i = 1; i < m; i++)
            bs[i] += bs[i - 1];
        for (i = n - 1; i >= 0; i--)
```

```
39              sa[--bs[wv[i]]] = y[i];
40          for (t = x, x = y, y = t, p = 1, x[sa[0]] = 0, i = 1; i < n; i++)
41              x[sa[i]] = cmp(y, sa[i - 1], sa[i], j) ? p - 1 : p++;
42      }
43      // 比原模板多出这两行是为了将下标从0..n-1映射为1..n。i<n是因为形参传的长度比实际长度大1（为了满足r[n
        +1]=0）
44      for (i = 1; i < n; ++i)
45          ++sa[i];
46      // 预处理rank
47      for (i = 1; i < n; ++i)
48          rk[sa[i]] = i;
49      return;
50  }
51
52  // O(n)求height数组
53  void getHeight(int r[], int sa[], int rk[], int ht[], int n) {
54      int i, k;
55      // 预处理height
56      for (i = 1, k = 0; i <= n; ++i) {
57          k -= k > 0;                                  // ht[rk[i]] >= ht[rk[i]-1] - 1 且 ht[] >= 0
58          while (r[i + k] == r[sa[rk[i] - 1] + k])  // 暴力求lcp
59              ++k;
60          ht[rk[i]] = k;
61      }
62  }
63
64  int main() {
65      int i, mx;  // mx是r[]中的最大值
66      cin >> s + 1;
67      n = strlen(s + 1);
68
69      // 关键就是这几行
70      for (i = 1; i <= n; ++i)
71          r[i] = s[i], mx = max(mx, r[i]);
72      r[n + 1] = 0;                    // 规定r[n+1]为0
73      da(r + 1, sa, n + 1, mx + 1);  // 这里注意是n+1,mx+1。填r+1是为了在接下来的处理中可以当成下标从1开
        始
74
75      // 下面下标统一为 r[1..n]  sa[1..n]=[1..n]  rk[1..n]=[1..n]   (与原模板一致)
76
77      getHeight(r, sa, rk, ht, n);
78
79      for (i = 1; i <= n; ++i)
80          cout << sa[i] << " ";
81      cout << endl;
82      for (i = 1; i <= n; ++i)
83          cout << rk[i] << " ";
84      cout << endl;
85      for (i = 1; i <= n; ++i)
86          cout << ht[i] << " ";
87      cout << endl;
88
89      return 0;
90  }
91
92  /*
93  aabaaaab
94  4 6 8 1 2 3 5 7
95  4 5 6 1 7 2 8 3
96  0 3 2 3 1 2 0 1
97  */
```