前缀函数与 KMP 算法

前缀函数定义

给定一个长度为 n 的字符串 s，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 $s[0 \dots i]$ 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，$\pi[0] = 0$。

前缀函数的定义可用数学语言描述如下：

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

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 $[0, 1, 0, 1, 2, 2, 3]$。

朴素算法

一个直接按照定义计算前缀函数的算法如下：

```
vector<int> prefix_function(string s) {
  int n = (int)s.length();
  vector<int> pi(n);
  for (int i = 0; i < n; i++)
    for (int k = 0; k <= i; k++)
      if (s.substr(0, k) == s.substr(i - k + 1, k)) pi[i] = k;
  return pi;
}
```

显见该算法的时间复杂度为 $O(n^3)$，具有很大的改进空间。

高效算法

该算法由 Knuth 和 Pratt 在 1977 年提出，同年 Morris 也独立的提出该算法。该算法被用作一个子串搜索算法的核心函数。

第一个优化

第一个重要的观察是相邻的前缀函数值至多增加 1。

实际上，如不然，即 $\pi[i+1] > \pi[i]+1$，考察长度为 $\pi[i+1]$ 的 $s[0 \dots i+1]$ 的后缀可引出矛盾。该后缀去掉最后一个字符后，我们得到一个长度为 $\pi[i+1]-1$ 的 $s[0 \dots i]$ 的后缀。该后缀比 $\pi[i]$ 描述的后缀更优，同其定义矛盾。

下述图例展示了这个矛盾。假定位于位置 i 和 $i+1$ 的既是后缀同时也是前缀的最长真后缀的长度分别为 2 和 4。则字符串 $s_0 s_1 s_2 s_3$ 与字符串 $s_{i-2} s_{i-1} s_i s_{i+1}$ 相同，这意味着 $s_0 s_1 s_2$ 与字符串 $s_{i-2} s_{i-1} s_i$ 相同，因此 $\pi[i]$ 至少为 3。

$$\overbrace{\underbrace{s_0\ s_1}_{\pi[i]=2}\ s_2\ s_3}^{\pi[i+1]=4} \ \dots\ \overbrace{s_{i-2}\ \underbrace{s_{i-1}\ s_i}_{\pi[i]=2}\ s_{i+1}}^{\pi[i+1]=4}$$

所以当移动到下一个位置时，前缀函数的值要么增加一，要么维持不变，要么减少。实际上，该事实已经允许我们将算法时间复杂度减少至 $O(n^2)$。因为每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。这意味着我们仅需要进行 $O(n)$ 次字符串比较，所以总复杂度为 $O(n^2)$。

第二个优化

让我们走的更远一点：尝试摆脱掉字符串比较。为了达成这一点，我们必须用到先前计算的所有信息。

现在考虑计算位置 $i+1$ 的前缀函数 π 的值。如果 $s[i+1] = s[\pi[i]]$，那么我们可以断言 $\pi[i+1] = \pi[i]+1$，因为我们已经知道位于位置 i 的长度为 $\pi[i]$ 的后缀同长度为 $\pi[i]$ 的前缀相等。参照下述图例：

$$\overbrace{\underbrace{s_0\ s_1\ s_2}_{\pi[i]}\ \underbrace{s_3}_{s_3=s_{i+1}}}^{\pi[i+1]=\pi[i]+1} \ \dots\ \overbrace{\underbrace{s_{i-2}\ s_{i-1}\ s_i}_{\pi[i]}\ \underbrace{s_{i+1}}_{s_3=s_{i+1}}}^{\pi[i+1]=\pi[i]+1}$$

如果不是上述情况，即 $s[i+1] \neq s[\pi[i]]$，那么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 $s[0 \dots j-1] = s[i-j+1 \dots i]$：

$$\overbrace{\underbrace{s_0\ s_1}_{j}\ s_2\ s_3}^{\pi[i]} \ \dots\ \overbrace{s_{i-3}\ s_{i-2}\ \underbrace{s_{i-1}\ s_i}_{j}}^{\pi[i]}\ s_{i+1}$$

实际上，如果我们找到了这样的长度 j，那么我们仅需要再次比较 $s[i+1]$ 和 $s[j]$。如果他们相等，那么我们置 $\pi[i+1] = j+1$。否则，我们需要找到小于 j 的最大值使得前缀性质得以保持，如此反复。这个过程会一直持续，直到 $j=0$。如果 $s[i+1] = s[0]$，那么我们置 $\pi[i+1] = 1$，否则 $\pi[i+1] = 0$。

所以我们已经有了这个算法的一个大概雏形。现在仅剩的问题是对于 j，如何快速找到这样的长度。让我们重新叙述一遍：对于当前在位置 i 使得前缀性质得以保持的长度 j，也即

$s[0 \dots j-1] = s[i-j+1 \dots i]$，我们希望找到最大的 $k < j$，使得前缀性质仍得以保持。

$$\overbrace{\underbrace{s_0\ s_1}_{k}\ s_2\ s_3}^{j} \dots \overbrace{s_{i-3}\ s_{i-2}\ \underbrace{s_{i-1}\ s_i}_{k}}^{j}\ s_{i+1}$$

上图显示出 k 必定为 $\pi[j-1]$，而该值我们之前已经计算过了。

最终算法

所以最终我们可以构建一个不需要进行任何字符串比较，并且只进行 $O(n)$ 次操作的算法。

以下是最终的流程：

- 在一个循环中以 $i = 1$ 到 $i = n - 1$ 的顺序计算前缀函数 $\pi[i]$ 的值（$\pi[0]$ 被赋值为 0）。
- 为了计算当前的前缀函数值 $\pi[i]$，我们令变量 j 表示右端点位于 $i-1$ 的最好的后缀的长度。初始时 $j = \pi[i-1]$。
- 通过比较 $s[j]$ 和 $s[i]$ 来检查长度为 $j+1$ 的后缀是否同时也是一个前缀。如果二者相等，那么我们置 $\pi[i] = j+1$，否则我们减少 j 至 $\pi[j-1]$ 并且重复该过程。
- 如果 $j = 0$ 并且仍没有任何一次匹配，则置 $\pi[i] = 0$ 并移至下一个下标 $i+1$。

实现

该算法的实现出人意料的短且直观。

```
vector<int> prefix_function(string s) {
  int n = (int)s.length();
  vector<int> pi(n);
  for (int i = 1; i < n; i++) {
    int j = pi[i - 1];
    while (j > 0 && s[i] != s[j]) j = pi[j - 1];
    if (s[i] == s[j]) j++;
    pi[i] = j;
  }
  return pi;
}
```

这是一个 **在线** 算法，即其当数据到达时处理它——举例来说，你可以一个字符一个字符的读取字符串，立即处理它们以计算出每个字符的前缀函数值。该算法仍然需要存储字符串本身以及先前计算过的前缀函数值，但如果我们已经预先知道该字符串前缀函数的最大可能取值 M，那么我们仅需要存储该字符串的前 $M+1$ 个字符以及对应的前缀函数值。

应用

在字符串中查找子串：Knuth-Morris-Pratt 算法

该任务是前缀函数的一个典型应用。

给定一个文本 t 和一个字符串 s，我们尝试找到并展示 s 在 t 中的所有出现（occurrence）。

为了简便起见，我们用 n 表示字符串 s 的长度，用 m 表示文本 t 的长度。

我们构造一个字符串 $s + \# + t$，其中 $\#$ 为一个既不出现在 s 中也不出现在 t 中的分隔符。接下来计算该字符串的前缀函数。现在考虑该前缀函数除去最开始 $n + 1$ 个值（即属于字符串 s 和分隔符的函数值）后其余函数值的意义。根据定义，$\pi[i]$ 为右端点在 i 且同时为一个前缀的最长真子串的长度，具体到我们的这种情况下，其值为与 s 的前缀相同且右端点位于 i 的最长子串的长度。由于分隔符的存在，该长度不可能超过 n。而如果等式 $\pi[i] = n$ 成立，则意味着 s 完整出现在该位置（即其右端点位于位置 i）。注意该位置的下标是对字符串 $s + \# + t$ 而言的。

因此如果在某一位置 i 有 $\pi[i] = n$ 成立，则字符串 s 在字符串 t 的 $i - (n - 1) - (n + 1) = i - 2n$ 处出现。

正如在前缀函数的计算中已经提到的那样，如果我们知道前缀函数的值永远不超过一特定值，那么我们不需要存储整个字符串以及整个前缀函数，而只需要二者开头的一部分。在我们这种情况下这意味着只需要存储字符串 $s + \#$ 以及相应的前缀函数值即可。我们可以一次读入字符串 t 的一个字符并计算当前位置的前缀函数值。

因此 Knuth-Morris-Pratt 算法（简称 KMP 算法）用 $O(n + m)$ 的时间以及 $O(n)$ 的内存解决了该问题。

统计每个前缀的出现次数

在该节我们将同时讨论两个问题。给定一个长度为 n 的字符串 s，在问题的第一个变种中我们希望统计每个前缀 $s[0 \dots i]$ 在同一个字符串的出现次数，在问题的第二个变种中我们希望统计每个前缀 $s[0 \dots i]$ 在另一个给定字符串 t 中的出现次数。

首先让我们来解决第一个问题。考虑位置 i 的前缀函数值 $\pi[i]$。根据定义，其意味着字符串 s 一个长度为 $\pi[i]$ 的前缀在位置 i 出现并以 i 为右端点，同时不存在一个更长的前缀满足前述定义。与此同时，更短的前缀可能以该位置为右端点。容易看出，我们遇到了在计算前缀函数时已经回答过的问题：给定一个长度为 j 的前缀，同时其也是一个右端点位于 i 的后缀，下一个更小的前缀长度 $k < j$ 是多少？该长度的前缀需同时也是一个右端点为 i 的后缀。因此以位置 i 为右端点，有长度为 $\pi[i]$ 的前缀，有长度为 $\pi[\pi[i] - 1]$ 的前缀，有长度为 $\pi[\pi[\pi[i] - 1] - 1]$ 的前缀，等等，直到长度变为 0。故而我们可以通过下述方式计算答案。

```
vector<int> ans(n + 1);
for (int i = 0; i < n; i++) ans[pi[i]]++;
for (int i = n - 1; i > 0; i--) ans[pi[i - 1]] += ans[i];
for (int i = 0; i <= n; i++) ans[i]++;
```

在上述代码中我们首先统计每个前缀函数值在数组 π 中出现了多少次，然后再计算最后答案：如果我们知道长度为 i 的前缀出现了恰好 $\mathbf{ans}[i]$ 次，那么该值必须被叠加至其最长的既是后缀也是前缀的子串的出现次数中。在最后，为了统计原始的前缀，我们对每个结果加 1 。

现在考虑第二个问题。我们应用来自 Knuth-Morris-Pratt 的技巧：构造一个字符串 $s + \# + t$ 并计算其前缀函数。与第一个问题唯一的不同之处在于，我们只关心与字符串 t 相关的前缀函数值，即 $i \geq n + 1$ 的 $\pi[i]$ 。有了这些值之后，我们可以同样应用在第一个问题中的算法来解决该问题。

一个字符串中本质不同子串的数目

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。

我们将迭代的解决该问题。换句话说，在知道了当前的本质不同子串的数目的情况下，我们要找出一种在 s 末尾添加一个字符后重新计算该数目的方法。

令 k 为当前 s 的本质不同子串数量。我们添加一个新的字符 c 至 s 。显然，会有一些新的子串以字符 c 结尾。我们希望对这些以该字符结尾且我们之前未曾遇到的子串计数。

构造字符串 $t = s + c$ 并将其反转得到字符串 $t^\sim$ 。现在我们的任务变为计算有多少 $t^\sim$ 的前缀未在 $t^\sim$ 的其余任何地方出现。如果我们计算了 $t^\sim$ 的前缀函数最大值 $\pi_{\max}$ ，那么最长的出现在 s 中的前缀其长度为 $\pi_{\max}$ 。自然的，所有更短的前缀也出现了。

因此，当添加了一个新字符后新出现的子串数目为 $|s| + 1 - \pi_{\max}$ 。

所以对于每个添加的字符，我们可以在 $O(n)$ 的时间内计算新子串的数目，故最终复杂度为 $O(n^2)$ 。

值得注意的是，我们也可以重新计算在头部添加一个字符，或者从尾或者头移除一个字符时的本质不同子串数目。

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。知道了该长度，该问题的答案即为长度为该值的 s 的前缀。

让我们计算 s 的前缀函数。通过使用该函数的最后一个值 $\pi[n-1]$ ，我们定义值 $k = n - \pi[n-1]$ 。我们将证明，如果 k 整除 n ，那么 k 就是答案，否则不存在一个有效的压缩，故答案为 n 。

假定 n 可被 k 整除。那么字符串可被划分为长度为 k 的若干块。根据前缀函数的定义，该字符串长度为 $n - k$ 的前缀等于其后缀。但是这意味着最后一个块同倒数第二个块相等，并且倒数第二

个块同倒数第三个块相等，等等。作为其结果，所有块都是相等的，因此我们可以将字符串 s 压缩至长度 k 。

诚然，我们仍需证明该值为最优解。实际上，如果有一个比 k 更小的压缩表示，那么前缀函数的最后一个值 $\pi[n-1]$ 必定比 $n-k$ 要大。因此 k 就是答案。

现在假设 n 不可以被 k 整除，我们将通过反证法证明这意味着答案为 n [1 [#fn:1]]。假设其最小压缩表示 r 的长度为 p （ p 整除 n ），字符串 s 被划分为 $n/p \geq 2$ 块。那么前缀函数的最后一个值 $\pi[n-1]$ 必定大于 $n-p$ （如果等于则 n 可被 k 整除），也即其所表示的后缀将部分的覆盖第一个块。现在考虑字符串的第二个块。该块有两种解释：第一种为 $r_0 r_1 \dots r_{p-1}$ ，另一种为 $r_{p-k} r_{p-k+1} \dots r_{p-1} r_0 r_1 \dots r_{p-k-1}$ 。由于两种解释对应同一个字符串，因此可得到 p 个方程组成的方程组，该方程组可简写为 $r_{(i+k) \bmod p} = r_{i \bmod p}$ ，其中 $\cdot \bmod p$ 表示模 p 意义下的最小非负剩余。

$$\overbrace{r_0\ r_1\ r_2\ r_3\ r_4\ r_5}^{p}\overbrace{r_0\ r_1\ r_2\ r_3\ r_4 r_5}^{p}$$
$$r_0\ r_1\ r_2\ r_3\ \overbrace{r_0\ r_1\ r_2\ r_3\ r_4\ r_5}^{p}\ r_0\ r_1$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{\pi[11]=8}$$

根据扩展欧几里得算法我们可以得到一组 x 和 y 使得 $xk + yp = \gcd(k, p)$ 。通过与等式 $pk - kp = 0$ 适当叠加我们可以得到一组 $x' > 0$ 和 $y' < 0$ 使得 $x'k + y'p = \gcd(k, p)$ 。这意味着通过不断应用前述方程组中的方程我们可以得到新的方程组 $r_{(i+\gcd(k,p)) \bmod p} = r_{i \bmod p}$ 。

由于 $\gcd(k, p)$ 整除 p ，这意味着 $\gcd(k, p)$ 是 r 的一个周期。又因为 $\pi[n-1] > n-p$ ，故有 $n - \pi[n-1] = k < p$ ，所以 $\gcd(k, p)$ 是一个比 p 更小的 r 的周期。因此字符串 s 有一个长度为 $\gcd(k, p) < p$ 的压缩表示，同 p 的最小性矛盾。

综上所述，不存在一个长度小于 n 的压缩表示，因此答案为 n 。

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隔符将两个字符串拼接的新字符串。对于字符串 s 和 t 我们计算 $s + \# + t$ 的前缀函数。显然，因为 $\#$ 是一个分隔符，前缀函数值永远不会超过 $|s|$ 。因此我们只需要存储字符串 $s + \#$ 和其对应的前缀函数值，之后就可以动态计算对于之后所有字符的前缀函数值：

$$\underbrace{s_0\ s_1\ \dots\ s_{n-1}\ \#}_{\text{need to store}}\ \underbrace{t_0\ t_1\ \dots\ t_{m-1}}_{\text{do not need to store}}$$

实际上在这种情况下，知道 t 的下一个字符 c 以及之前位置的前缀函数值便足以计算下一个位置的前缀函数值，而不需要用到任何其它 t 的字符和对应的前缀函数值。

换句话说，我们可以构造一个 **自动机** （一个有限状态机）：其状态为当前的前缀函数值，而从一个状态到另一个状态的转移则由下一个字符确定。

因此，即使没有字符串 t ，我们同样可以应用构造转移表的算法构造一个转移表 $(\text{old } \pi, c) \to \text{new } \_\pi$ ：

```
void compute_automaton(string s, vector<vector<int>>& aut) {
  s += '#';
  int n = s.size();
  vector<int> pi = prefix_function(s);
  aut.assign(n, vector<int>(26));
  for (int i = 0; i < n; i++) {
    for (int c = 0; c < 26; c++) {
      int j = i;
      while (j > 0 && 'a' + c != s[j]) j = pi[j - 1];
      if ('a' + c == s[j]) j++;
      aut[i][c] = j;
    }
  }
}
```

然而在这种形式下，对于小写字母表，算法的时间复杂度为 $O(|\Sigma| n^2)$ 。注意到我们可以应用动态规划来利用表中已计算过的部分。只要我们从值 j 变化到 $\pi[j-1]$ ，那么我们实际上在说转移 (j, c) 所到达的状态同转移 $(\pi[j-1], c)$ 一样，但该答案我们之前已经精确计算过了。

```
void compute_automaton(string s, vector<vector<int>>& aut) {
  s += '#';
  int n = s.size();
  vector<int> pi = prefix_function(s);
  aut.assign(n, vector<int>(26));
  for (int i = 0; i < n; i++) {
    for (int c = 0; c < 26; c++) {
      if (i > 0 && 'a' + c != s[i])
        aut[i][c] = aut[pi[i - 1]][c];
      else
        aut[i][c] = i + ('a' + c == s[i]);
    }
  }
}
```

最终我们可在 $O(|\Sigma| n)$ 的时间复杂度内构造该自动机。

该自动机在什么时候有用呢？首先，记得大部分时候我们为了一个目的使用字符串 $s + \# + t$ 的前缀函数：寻找字符串 s 在字符串 t 中的所有出现。

因此使用该自动机的最直接的好处是 **加速计算字符串 $s + \# + t$ 的前缀函数** 。

通过构建 $s + \#$ 的自动机，我们不再需要存储字符串 s 以及其对应的前缀函数值。所有转移已经在表中计算过了。

但除此以外，还有第二个不那么直接的应用。我们可以在字符串 t 是 **某些通过一些规则构造的巨型字符串** 时，使用该自动机加速计算。Gray 字符串，或者一个由一些短的输入串的递归组合所构

造的字符串都是这种例子。

出于完整性考虑，我们来解决这样一个问题：给定一个数 $k \leq 10^5$ ，以及一个长度 $\leq 10^5$ 的字符串 s ，我们需要计算 s 在第 k 个 Gray 字符串中的出现次数。回想起 Gray 字符串以下述方式定义：

$$
\begin{aligned}
g_1 &= \mathbf{a} \\
g_2 &= \mathbf{aba} \\
g_3 &= \mathbf{abacaba} \\
g_4 &= \mathbf{abacabadabacaba}
\end{aligned}
$$

由于其天文数字般的长度，在这种情况下即使构造字符串 t 都是不可能的：第 k 个 Gray 字符串有 $2^k - 1$ 个字符。然而我们可以在仅仅知道开头若干前缀函数值的情况下，有效计算该字符串末尾的前缀函数值。

除了自动机之外，我们同时需要计算值 $G[i][j]$ ：在从状态 j 开始处理 g_i 后的自动机的状态，以及值 $K[i][j]$ ：当从状态 j 开始处理 g_i 后， s 在 g_i 中的出现次数。实际上 $K[i][j]$ 为在执行操作时前缀函数取值为 $|s|$ 的次数。易得问题的答案为 $K[k][0]$ 。

我们该如何计算这些值呢？首先根据定义，初始条件为 $G[0][j] = j$ 以及 $K[0][j] = 0$ 。之后所有值可以通过先前的值以及使用自动机计算得到。为了对某个 i 计算相应值，回想起字符串 g_i 由 g_{i-1} ，字母表中第 i 个字符，以及 g_{i-1} 三者拼接而成。因此自动机会途径下列状态：

$$
\begin{aligned}
\mathrm{mid} &= \mathrm{aut}[G[i-1][j]][i] \\
G[i][j] &= G[i-1][\mathrm{mid}]
\end{aligned}
$$

$K[i][j]$ 的值同样可被简单计算。

$$
K[i][j] = K[i-1][j] + [\mathrm{mid} == |s|] + K[i-1][\mathrm{mid}]
$$

其中 $[\cdot]$ 当其中表达式取值为真时值为 1 ，否则为 0 。综上，我们已经可以解决关于 Gray 字符串的问题，以及一大类与之类似的问题。举例来说，应用同样的方法可以解决下列问题：给定一个字符串 s 以及一些模式 t_i ，其中每个模式以下列方式给出：该模式由普通字符组成，当中可能以 t_k^{cnt} 的形式递归插入先前的字符串，也即在该位置我们必须插入字符串 t_k cnt 次。以下是这些模式的一个例子：

$$
\begin{aligned}
t_1 &= \mathbf{abdeca} \\
t_2 &= \mathbf{abc} + t_1^{30} + \mathbf{abd} \\
t_3 &= t_2^{50} + t_1^{100} \\
t_4 &= t_2^{10} + t_3^{100}
\end{aligned}
$$

递归代入会使字符串长度爆炸式增长，他们的长度甚至可以达到 100^{100} 的数量级。而我们必须找到字符串 s 在每个字符串中的出现次数。

该问题同样可通过构造前缀函数的自动机解决。同之前一样，我们利用先前计算过的结果对每个模式计算其转移然后相应统计答案即可。