# 前缀函数与 KMP 算法

## 字符串前缀和后缀定义

关于字符串前缀、真前缀，后缀、真后缀的定义详见

**字符串基础**

## 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 $\pi$，其中 $\pi[i]$ 的定义是：

1. 如果子串 $s[0\cdots i]$ 有一对相等的真前缀与真后缀（$s[0\ldots k-1]$ 和 $s[i-(k-1)\ldots i]$），那么 $\pi[i]$ 就是这个相等的真前缀（或者真后缀，因为它们相等）的长度，也就是 $\pi[i]=k$；
2. 如果不止有一对相等的，那么 $\pi[i]$ 就是其中最长的那一对的长度；
3. 如果没有相等的，那么 $\pi[i]=0$。

简单来说 $\pi[i]$ 就是，子串 $s[0\ldots i]$ 最长的相等的真前缀与真后缀的长度。

用数学语言描述如下：$$ \pi[i]=\max_{k=0\ldots i}\{k:s[0\ldots k-1]=s[i-(k-1)\ldots i]\} $$ 特别地，规定 $\pi[0]=0$。

举例来说，对于字符串abcabcd：

$\pi[0]=0$，因为a没有真前缀和真后缀，根据规定为 $0$

$\pi[1]=0$，因为ab无相等的真前缀和真后缀

$\pi[2]=0$，因为abc无相等的真前缀和真后缀

$\pi[3]=1$，因为abca只有一对相等的真前缀和真后缀：a，长度为 $1$

$\pi[4]=2$，因为abcab相等的真前缀和真后缀为ab，长度为 $2$

$\pi[5]=3$，因为abcabc相等的真前缀和真后缀为abc，长度为 $3$

$\pi[6]=0$，因为abcabcd无相等的真前缀和真后缀

同理可以计算字符串 aabaaab 的前缀函数为 $[0,1,0,1,2,2,3]$。

## 计算前缀函数的朴素算法

一个直接按照定义计算前缀函数的算法流程：

- 在一个循环中以 $i=1\rightarrow n-1$ 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 $j$ 从最大的真前缀长度 $i$ 开始尝试。

- 如果当前长度下真前缀和真后缀相等，则此时长度为 $\pi[i]$否则令 $j$ 自减 $1$，继续匹配，直到 $j=0$
- 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 $i+1$

具体实现如下：

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

注：

- `string substr (size_t pos = 0, size_t len = npos) const;`

显见该算法的时间复杂度为 $O(n^3)$具有很大的改进空间。

# 计算前缀函数的高效算法

## 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 $1$**

参照下图所示，只需如此考虑：当取一个尽可能大的 $\pi[i+1]$ 时，必然要求新增的 $s[i+1]$ 也与之对应的字符匹配，即 $s[i+1]=s[\pi[i]]$此时 $\pi[i+1]=\pi[i]+1$ $$ \underbrace{\overbrace{s_0~s_1~s_2}^{\pi[i]=3}~s_3}_{\pi[i+1]=4}~\cdots~\underbrace{\overbrace{s_{i-2}~s_{i-1}~s_{i}}^{\pi[i]=3}~s_{i+1}}_{\pi[i+1]=4} $$ 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。

此时的改进的算法为：

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

```
}
```

在这个初步改进的算法中，在计算每个 $\pi[i]$ 时，最好的情况是第一次字符串比较就完成了匹配，也就是说基础的字符串比较次数是 n-1 次。

而由于存在 j=pi[i-1]+1（pi[0]=0）对于最大字符串比较次数的限制，可以看出每次只有在最好情况才会为字符串比较次数的上限积累 $1$，而每次超过一次的字符串比较消耗的是之后次数的增长空间。

由此我们可以得出字符串比较次数最多的一种情况：至少 1 次字符串比较次数的消耗和最多 n-2 次比较次数的积累，此时字符串比较次数为 n-1 + n-2 = 2n-3。

可见经过此次优化，计算前缀函数只需要进行 $O(n)$ 次字符串比较，总复杂度降为了 $O(n^2)$。

## 第二个优化

在第一个优化中，我们讨论了计算 $\pi[i+1]$ 时的最好情况：$s[i+1]=s[\pi[i]]$，此时 $\pi[i+1]=\pi[i]+1$。现在让我们沿着这个思路走得更远一点：讨论当 $s[i+1]\ne s[\pi[i]]$ 时如何跳转。

失配时，我们希望找到对于子串 $s[0\ldots i]$，仅次于 $\pi[i]$ 的第二长度 $j$，使得在位置 $i$ 的前缀性质仍得以保持，也即 $s[0\ldots j-1]=s[i-j+1\ldots i]$。 $$ \overbrace{\underbrace{s_0 ~ s_1}_j ~ s_2 ~ s_3}^{\pi[i]} ~ \dots ~ \overbrace{s_{i-3} ~ s_{i-2} ~ \underbrace{s_{i-1} ~ s_{i}}_j}^{\pi[i]} ~ s_{i+1} $$ 如果我们找到了这样的长度 $j$，那么仅需要再次比较 $s[i+1]$ 和 $s[j]$，如果它们相等，那么就有 $\pi[i+1]=j+1$。否则，我们需要找到子串 $s[0\ldots i]$ 仅次于 $j$ 的第二长度 $j^{(2)}$，使得前缀性质得以保持，如此反复，直到 $j=0$。如果 $s[i+1]\ne s[0]$，则 $\pi[i+1]=0$。

观察上图可以发现，因为 $s[0\ldots \pi[i]]=s[i-\pi[i]+1\ldots i]$，所以对于 $s[0\ldots i]$ 的第二长度 $j$，有这样的性质： $$ s[0\ldots j-1]=s[i-j+1\ldots i]=s[\pi[i]-j\ldots \pi[i]-1] $$ 也就是说 $j$ 等价于子串 $s[\pi[i]-1]$ 的前缀函数值，即 $j=\pi[\pi[i]-1]$。同理，次于 $j$ 的第二长度等价于 $s[j-1]$ 的前缀函数值：$j^{(2)}=\pi[j-1]$

显然我们可以得到一个关于 $j$ 的状态转移方程：$j^{(n)}=\pi[j^{(n-1)}-1]$，$(j^{n-1}>0)$

## 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。

而且该算法的实现出人意料地短且直观：

```
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地读取字符串，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。

*吐槽一下*：虽然这个改进后的计算前缀函数的算法看起来很厉害，但是在基准测试中发现，当模式串 $s$ 的长度 $n$ 不是很大（100以内）的情况下，其实和朴素算法也没有什么区别。

# 应用

## 在字符串中查找子串Knuth-Morris-Pratt 算法

该算法由 KnuthPratt 和 Morris 在 1977 年共同发布。

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的值，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 $s[0\ldots i]$ 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 $s[0\ldots i]$ 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 $\pi$ 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 $ans[i]$ 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 $i\ge n+1$ 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

待补

### 字符串压缩

待补

### 根据前缀函数构建一个自动机

待补

## 练习题目

- UVA 455 “Periodic Strings”
- UVA 11022 “String Factoring”
- UVA 11452 “Dancing the Cheeky-Cheeky”
- UVA 12604 - Caesar Cipher
- UVA 12467 - Secret Word
- UVA 11019 - Matrix Matcher
- SPOJ - Pattern Find
- Codeforces - Anthem of Berland
- Codeforces - MUH and Cube Walls

## 参考链接

Oi Wiki