字符串算法

张敬东

福州第十六中学

2023 年 11 月 26 日

目录

一些定义

KMP

exKMP/Z 函数

Manacher

Trie

目录

一些定义

KMP

exKMP/Z 函数

Manacher

Trie

一些定义

- 默认字符串下标从 0 开始。

一些定义

- 默认字符串下标从 0 开始。
- 字符串 s 中字符的个数称为**字符串的长度，记作** $|s|$；

一些定义

- 默认字符串下标从 0 开始。
- 字符串 s 中字符的个数称为**字符串的长度，记作** $|s|$；
- s 中第 i 个字符记作 s_i。

一些定义

- 默认字符串下标从 0 开始。
- 字符串 s 中字符的个数称为**字符串的长度，记作** $|s|$；
- s 中第 i 个字符记作 s_i。
- s 的子串 $s[i \ldots j]$ 表示字符 $s[i], \ldots, s[j]$ 串起来形成的字符串；

一些定义

- 默认字符串下标从 0 开始。
- 字符串 s 中字符的个数称为**字符串的长度，记作** $|s|$；
- s 中第 i 个字符记作 s_i。
- s 的子串 $s[i \dots j]$ 表示字符 $s[i], \dots, s[j]$ 串起来形成的字符串；
- $s[1 \dots i]$ 是 s 到 i 的前缀，$s[i \dots |s|-1]$ 是 s 的后缀。
  其中若 $s[1 \dots i] \neq s$，则称其为 s 的真前缀。真后缀同理。

一些定义

- 默认字符串下标从 0 开始。
- 字符串 s 中字符的个数称为**字符串的长度，记作** $|s|$；
- s 中第 i 个字符记作 s_i。
- s 的子串 $s[i \dots j]$ 表示字符 $s[i], \dots, s[j]$ 串起来形成的字符串；
- $s[1 \dots i]$ 是 s 到 i 的前缀，$s[i \dots |s|-1]$ 是 s 的后缀。
  其中若 $s[1 \dots i] \neq s$，则称其为 s 的真前缀。真后缀同理。
- 回文串是形如 abcba, abccba 的字符串。

目录

一些定义

KMP

exKMP/Z 函数

Manacher

Trie

字符串查找

有时我们希望在文本串 t 中查找模式串 s。比如你按下 Ctrl+F 时浏览器就会在页面中查找你输入的字符串。

字符串查找

有时我们希望在文本串 t 中查找模式串 s。比如你按下 Ctrl+F 时浏览器就会在页面中查找你输入的字符串。
一种方法是暴力查找，时间复杂度 $O(nm)$。有没有更快的方法呢？

字符串查找

有时我们希望在文本串 t 中查找模式串 s。比如你按下 Ctrl+F 时浏览器就会在页面中查找你输入的字符串。
一种方法是暴力查找，时间复杂度 $O(nm)$。有没有更快的方法呢？

注意

本节中规定：文本串 t，模式串 s，$m = |t|, n = |s|$。

前缀函数

定义：

- s 的 border：s 的最长公共真前后缀。例如 `abccdabc` 的 border 为 `abc`。
- s 前缀函数 π_i：$s[1\dots i]$ 的 border 长度。显然 $\pi_0=0$。

前缀函数

定义：

- s 的 border： s 的最长公共真前后缀。例如 `abccdabc` 的 border 为 `abc`。
- s 前缀函数 π_i： $s[1\dots i]$ 的 border 长度。显然 $\pi_0 = 0$。

对于 $s =$ `abcabcd`，

$\pi_0 = 0, \pi_1 = 0, \pi_2 = 0, \pi_3 = 1, \pi_4 = 2, \pi_5 = 3, \pi_6 = 0$。

前缀函数

定义：

- s 的 border： s 的最长公共真前后缀。例如 `abccdabc` 的 border 为 `abc`。
- s 前缀函数 π_i： $s[1 \dots i]$ 的 border 长度。显然 $\pi_0 = 0$。

对于 $s =$ `abcabcd`，
$\pi_0 = 0, \pi_1 = 0, \pi_2 = 0, \pi_3 = 1, \pi_4 = 2, \pi_5 = 3, \pi_6 = 0$。
那么如何计算前缀函数呢？

前缀函数的计算

观察可以发现 π_i 至多增加 1，也就是说如果考虑现在算 π_i，那么我们希望最好 $\pi_i = \pi_{i-1} + 1$，此时 $s[\pi_{i-1}] = s[i]$。

前缀函数的计算

观察可以发现 π_i 至多增加 1，也就是说如果考虑现在算 π_i，那么我们希望最好 $\pi_i = \pi_{i-1} + 1$，此时 $s[\pi_{i-1}] = s[i]$。
如：

$$\underbrace{\overbrace{s_0\ s_1\ s_2}^{\pi_{i-1}=3}\ s_3}_{\pi_i=4}\ \dots\ \underbrace{\overbrace{s_{i-3}\ s_{i-2}\ s_{i-1}}^{\pi_{i-1}=3}\ s_i}_{\pi_i=4}$$

（来自 OI-Wiki。）
此时 $\pi_{i-1} = 3$，我们希望 $s[3] = s[i]$，这样就可以得到 $\pi_i = \pi_{i-1} + 1$。

但是，如果 $s[\pi_{i-1}] \neq s[i]$，那怎么办？我们将这种情况称为“失配”。

但是，如果 $s[\pi_{i-1}] \neq s[i]$，那怎么办？我们将这种情况称为“失配”。

失配时，我们依然想让 border 尽可能地长，所以我们希望找到一个 **严格次长 border**。我们将原 border 记为 b，严格次长 border 记为 b'。

但是，如果 $s[\pi_{i-1}] \neq s[i]$，那怎么办？我们将这种情况称为“失配”。
失配时，我们依然想让 border 尽可能地长，所以我们希望找到一个 **严格次长 border**。我们将原 border 记为 b，严格次长 border 记为 b'。

1. 首先 $|b| > |b'|$（所以是“真前后缀”）。

但是，如果 $s[\pi_{i-1}] \neq s[i]$，那怎么办？我们将这种情况称为“失配”。

失配时，我们依然想让 border 尽可能地长，所以我们希望找到一个 **严格次长 border**。我们将原 border 记为 b，严格次长 border 记为 b'。

1. 首先 $|b| > |b'|$（所以是“真前后缀”）。
2. 然后又由于 b' 和 b 均是 $s[1 \ldots i-1]$ 的公共真前后缀，故 b' 是 b 的公共真前后缀（可以从下图看出）。**注意，b 和 b' 并不表示长度，而是字符串**，左右两对字符串 b 和 b' 是分别相等的。左边可以看出 b' 是 b 的前缀，右边可以看出 b' 是 b 的后缀。

但是，如果 $s[\pi_{i-1}] \neq s[i]$，那怎么办？我们将这种情况称为“失配”。

失配时，我们依然想让 border 尽可能地长，所以我们希望找到一个 **严格次长 border**。我们将原 border 记为 b，严格次长 border 记为 b'。

1. 首先 $|b| > |b'|$（所以是“真前后缀”）。
2. 然后又由于 b' 和 b 均是 $s[1 \dots i-1]$ 的公共真前后缀，故 b' 是 b 的公共真前后缀（可以从下图看出）。**注意，b 和 b' 并不表示长度，而是字符串**，左右两对字符串 b 和 b' 是分别相等的。左边可以看出 b' 是 b 的前缀，右边可以看出 b' 是 b 的后缀。
3. 因为我们规定是**严格次长 border**，所以除了 b 没有其他 border 比 b' 长，又因为 b' 是 b 的公共真前后缀，**所以 b' 是 b 的 border**。

$$\overbrace{\underbrace{s_0\ \ s_1}_{b'}\ \ s_2\ \ s_3}^{b}\ \ \cdots\ \ \overbrace{s_{i-4}\ \ s_{i-3}\ \ \underbrace{s_{i-2}\ \ s_{i-1}}_{b'}}^{b}\ \ s_i$$

总结：前缀函数

所以我们找到了一个递归关系：一个 border 的严格次长 border 为它自己的 border。（好绕啊！）前缀函数的求法就出来了：

- 先看是否满足 $s[\pi_{i-1}] = s[i]$。
- 如果满足，那么 $\pi_i = \pi_{i-1} + 1$。
- 否则就是失配，检查严格次长 border 能否匹配：$s[|b'|] = s[i]$。
  - 若还是不能，继续上一个操作，使 $b \leftarrow b', b' \leftarrow b$ border。（检查严格次长 border 的严格次长 border，或者说检查严格次次长 border）。直到无解或匹配上。
  - 如果匹配，$\pi_i = |b'|$。
  - 如果无解，$\pi_i = 0$。

代码：前缀函数

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的短且直观：（摘自 OI-Wiki。）

```
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;
}
```

KMP

说了半天前缀函数，KMP 到底怎么回事？

KMP

说了半天前缀函数，KMP 到底怎么回事？
你先别急，你如果能真的理解前缀函数，那 KMP 就是小菜一碟了。

KMP

说了半天前缀函数，KMP 到底怎么回事？
你先别急，你如果能真的理解前缀函数，那 KMP 就是小菜一碟了。
在文本串 t 中查找模式串 s 时，我们令 $u = s + \# + t$，其中 $+$ 为拼接，$\#$ 为任意 s 和 t 中不包含的字符。对 u 求前缀函数，若 $\pi_i = n$，那么 $u[i - n + 1 \ldots i] = s$。

KMP

说了半天前缀函数，KMP 到底怎么回事？
你先别急，你如果能真的理解前缀函数，那 KMP 就是小菜一碟了。
在文本串 t 中查找模式串 s 时，我们令 $u = s + \# + t$，其中 $+$ 为拼接，$\#$ 为任意 s 和 t 中不包含的字符。对 u 求前缀函数，若 $\pi_i = n$，那么 $u[i-n+1 \ldots i] = s$。
细想为什么：若 $\pi_i = n$，那么 $s[1 \ldots i]$ 的最长公共真前后缀就是 s，也就是说 t 中也出现了一个完整的 s，我们就成功在 t 中查找到 s 啦！

代码：KMP

```
vector<int> find_occurrences(string t, string s) {
    string cur = s + '#' + t;
    int m = t.size(), n = s.size();
    vector<int> v;
    vector<int> lps = prefix_function(cur);
    for (int i = n + 1; i <= m + n; i++) {
        if (lps[i] == n)
            v.push_back(i - 2 * n);
    }
    return v;
}
```

目录

一些定义

KMP

exKMP/Z 函数

Manacher

Trie

an exKMP.

目录

一些定义

KMP

exKMP/Z 函数

Manacher

Trie

a Manacher.

目录

一些定义

KMP

exKMP/Z 函数

Manacher

Trie