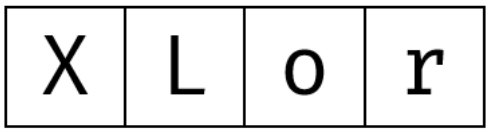
后缀结构

在现实生活中，字符串与人们密切相关，多彩的互联网，日常交流的语言，甚至 DNA 序列，本质都是字符串。

在计算机科学领域，字符串的地位更加重要，我们编写的代码都是一堆字符构成的线性序列，一个个小小的字符构筑起了计算机科学的大厦。

在算法竞赛中，字符串是一个有趣的专题。本文将着重介绍一系列关于字符串匹配问题的算法和数据结构，而这些算法和数据结构大多与字符串后缀的结构和性质紧密相关，因此标题定为 **后缀结构** (String Suffix Structure)。



字符串基础

形式定义

在正式讨论字符串之前，必须先规定何为字符。

定义字符集 (alphabet) 为一个非空的有限集合，通常记为 Σ，这个集合内的所有元素被称为字符 (letter / character)。字符集大小记为 $|\Sigma|$。

为了方便起见，在所有例题中除非特别说明，字符集均为 26 个小写字母。 当然，后文介绍的部分算法也可以解决更大字符集的问题，这将会具体讨论。

一个定义在 Σ 上的字符串 (string) 是一个由 0 个或多个属于 Σ 的字符组成的有限序列，记字符串 s 的长度为 $|s|$。

$$(s_1, s_2, \ldots, s_{|s|})$$

简记为 $s_1 s_2 \ldots s_{|s|}$。**注意本文均采用 1 作为数组或序列的起始下标。**

特别地，唯一一个长度等于 0 的字符串记为 ε 。

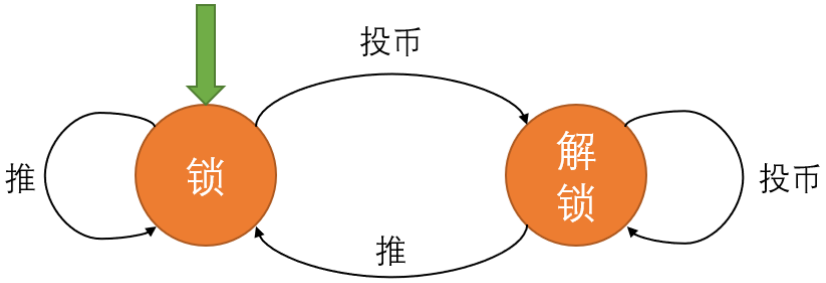
记定义在 Σ 上所有长度为 n 的字符串集合为 Σ^n， Σ 的克林闭包 Σ^* 为定义在 Σ 上的所有字符串，即 $\Sigma^* = \cup_{n=0}^{\infty} \Sigma^n$。例如 $\Sigma = \{0, 1\}$，则 $\Sigma^0 = \{\varepsilon\}$，$\Sigma^1 = \{0, 1\}$，$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, \ldots\}$。

字符串连接 (concatenation) 是定义在 Σ^* 上的二元运算，假设有字符串 s 和 t ，则 s 和 t 的连接结果为 $s_1 s_2 \ldots s_{|s|} t_1 t_2 \ldots t_{|t|}$，简记为 st。一个显而易见的结论，字符串连接满足结合律；简单推广，由于 ε 的存在， $(\Sigma^*, \text{concatenation})$ 是一个幺半群 (monoid)。

有限状态自动机

有限状态自动机 (Finite state automaton) 是一种计算模型 (model of computation)。它是一种抽象的机器，每个时刻这个机器会处于某种具体的状态。通过向这个发送输入内容，机器会在有限的状态之间转移 (transition)。

以🚇地铁进站口的闸门为例，一开始闸门处于🔒锁住的状态，你不管怎么推它都无法将它🔓解锁，当你向它💰投币时，闸门才会🔓解锁，这时你才可以推门进入，闸门恢复锁住的状态。



在算法竞赛中，我们使用较多的是 **确定性有限状态自动机** (deterministic finite state automaton)，下面给出它的形式化定义，确定性有限状态自动机被定义为一个五元组 $(\Sigma, S, s_0, \delta, F)$：

- Σ 是输入的字符集；
- S 是一个非空状态集合；
- s_0 是起始状态 $(s_0 \in S)$；
- δ 是一个状态转移函数，$\delta : S \times \Sigma \to \Sigma$；
- F 是终止状态集合 $F \subseteq S$。

子串

字符串的子串 (substring) 定义为该字符串中的某一段连续区间内字符所组成的字符串。形式化地，t 是 s 的子串，当且仅当存在 $u, v \in \Sigma^*$，满足 $utv = s$。字符串 s 的 $[l, r]$ $(1 \le l \le r \le |s|)$ 子串简记为 $s_{l \ldots r}$。

注意子串与子序列 (subsequence) 的区别，子序列只需要满足能够从原串中删除一些字符后，余下的字符按原来的相对顺序依次连接形成的字符串，而不一定是一段连续区间。形式化地，t 是 s 的子序列，当且仅当存在 $x_i \in \Sigma^* (1 \le i \le |t| + 1)$，满足 $x_1 t_1 x_2 t_2 \ldots x_{|t|} t_{|t|} x_{|t|+1}$。

字符串 s 的长度为 x 前缀 (prefix) 定义为由该字符串前 $x (0 \le x \le |s|)$ 个字符组成的子串，记为 $pre(s, x) = s_1 s_2 \ldots s_x$。类似地，长度为 x 的后缀 (suffix) 定义为由该字符串后 $x (0 \le x \le |s|)$ 个字符组成的子串，记为 $suf(s, x) = s_{|s|-x+1} s_{|s|-x+2} \ldots s_{|s|}$。真前缀和真后缀表示排除字符串本身的前缀或后缀。

字典序

两个字符串相等，当且仅当两个字符串长度相同且所有位置的字符对应相同。形式化地，$s = t$，当且仅当，$|s| = |t|$，$\forall 1 \leq i \leq |s|, s_i = t_i$。

通常会在字符集 Σ 上定义一个全序关系。例如，可以在字符集 $\Sigma = \{0, 1\}$ 上定义 $0 < 1$。有了字符集上的全序关系后，可以继续定义两个字符串的大小关系。

假设有两个不相等的字符串 s 和 t，$s < t$，当且仅当，s 是 t 的前缀，即 $s = pre(t, |s|)$，或者存在下标 $i (1 \leq i \leq |s|)$，$pre(s, i-1) = pre(t, i-1)$，且 $s_i < t_i$。

回文串

定义字符串 s 的翻转为 s^R，即从后往前读出这个字符串。形式化地，$s^R = s_{|s|} s_{|s|-1} \dots s_1$。

定义字符串 s 为回文串 (palindrome)，当且仅当 $s = s^R$。

实现

在计算机编程实现时，通常使用一个线性表来存储一个字符串，例如 `C` 风格的 `char` 类型数组，`C++` 风格的 `std::string`，以及例如 `JavaScript` 等高级编程语言标准库中提供的 `String` 或数组。

前置知识

> 本章节将列举出在解决后文例题中出现的基础算法或数据结构，由于它们与本文核心内容关系不大，故在此仅做粗略介绍，具体内容留给读者自行研究学习。

动态规划

动态规划（Dynamic programming，简称 DP）是一种在数学、管理科学、计算机科学、经济学和生物信息学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。

动态规划常常适用于有重叠子问题和最优子结构性质的问题，动态规划方法所耗时间往往远少于朴素解法。

动态规划背后的基本思想非常简单。大致上，若要解一个给定问题，我们需要解其不同部分（即子问题），再根据子问题的解以得出原问题的解。

通常许多子问题非常相似，为此动态规划法试图仅仅解决每个子问题一次，从而减少计算量：一旦某个给定子问题的解已经算出，则将其记忆化存储，以便下次需要同一个子问题解之时直接查表。这种做法在重复子问题的数目关于输入的规模呈指数增长时特别有用。

> 引用自 动态规划部分简介 - OI Wiki。

常见的基础动态规划问题：斐波那契数列，矩阵链乘，背包问题，最长上升子序列，最长公共子序列等。

图论

图论 (Graph theory) 是数学的一个分支，图是图论的主要研究对象。 图 (Graph) 是由若干给定的顶点及连接两顶点的边所构成的图形，这种图形通常用来描述某些事物之间的某种特定关系。顶点用于代表事物，连接两顶点的边则用于表示两个事物间具有这种关系。

> 引用自 图论部分简介 - OI Wiki。

本文中涉及的一些图论问题，大部分是由自动机转化来的问题，涉及一些图的存储和遍历，有向无环图的相关问题常见做法（例如 DAG 上最长路的动态规划算法）。

树论

在本文的算法中会出现一些与树相关结构，树是一种特殊图，相关的具体概念和基础知识可以参考 树基础 - OI Wiki。在例题中，涉及到树上差分，最近公共祖先，树和数据结构结合，相关内容和例题可以参考笔者 2019 年的算法竞赛专题报告 树论 | XLor's Blog。

字符串匹配自动机

问题 1

给定 1 个模板串 p 和 1 个文本串 s，判断 p 是否是 s 的子串，如果是的话，找到 p 在 s 中的所有出现位置。

其中 $1 \leq |p|, |s| \leq 10^6$。

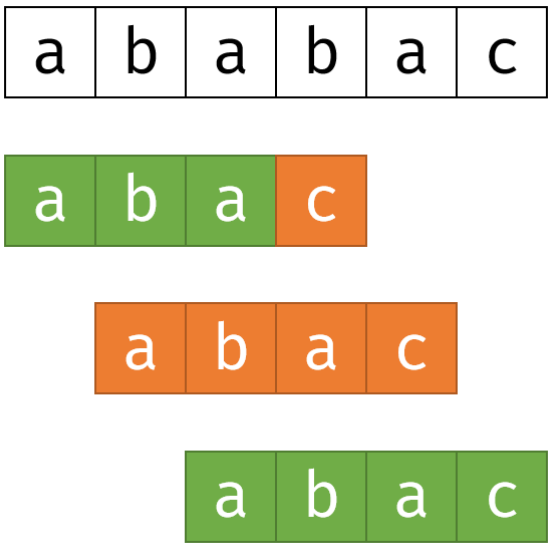
暴力

一个最简单的想法是，枚举 s 串的每个位置开始的长度为 $|p|$ 的子串，检查它是否与 p 相同，相同则输出答案，然后继续检查下一个位置，直到没有可能的位置。伪代码如下所示：

Input. Template string p, text string s
Output. All the occurence of p in t.
Method.

```
1  for i ← 1 to |s| − |p| + 1
2     flag ← true
3     for j ← 1 to |p|
4        if s_{i+j−1} ≠ p_j
5           then flag ← false, break
6     if flag then  find an occurence i
```

下图为 s 为 $ababac$，p 为 $abac$ 时暴力算法的运行示意 (绿色的位置表示字符相同，橙色的位置表示字符不同)。

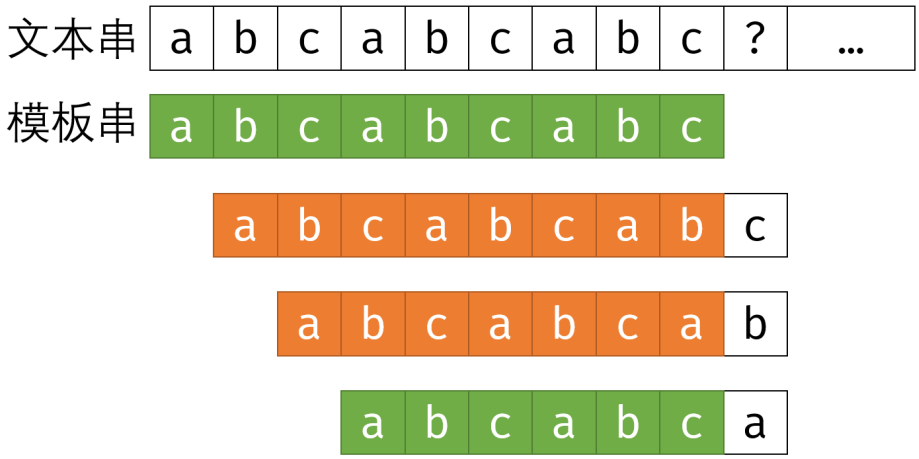


令 $n = |s|, m = |p|$，显而易见，暴力算法的时间复杂度为 $O(nm)$，空间复杂度为 $O(1)$，算法的时间效率极低，难以帮助我们解决问题一。

字符串匹配自动机

在上述暴力算法中，并没有有效利用到已经匹配好的信息（伪代码第 4 行）。

我们将暴力枚举的过程，想象成用一个长度为 $|p|$ 的窗口在文本串 s 上滑动。假如模板串是 $abcabcabc$，它已经在文本串中的某个位置匹配完成，我们现在需要向右滑动模板串的窗口。



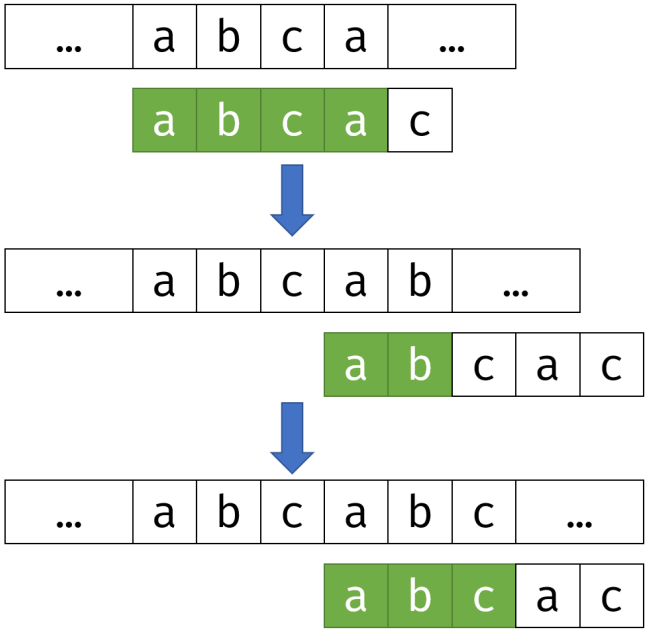
由上图我们容易发现，模板串向右滑动 1 格或者 2 格都是不可能匹配的。因为，这利用了已经在当前位置匹配好模板串 $abcabcabc$ 这一信息，滑动 1 格或者 2 格时，不可能和已确定的部分匹配。

我们继续滑动模板串，可以发现只有滑动次数在 3 格，6 格或 9 格（与已知部分不重叠，无法利用已知的信息）时才有可能在文本串发生匹配。深入本质，假如已经匹配了模板串 p，滑动格数 i 与已知不冲突，可能会在之后与文本串匹配，当且仅当 $pre(p, |p| - i) = suf(p, |p| - i)$，即 p 长度为 $|p| - i$ 的前后缀相同。

上面的结论对我们优化算法带来了启发，我们可以对于文本串的每个前缀，判断相应的后缀是否与模板串匹配，然后利用结论大步地滑动模板串。更进一步，引入后缀函数 $\sigma : \Sigma^* \to \{0, 1, \ldots, |p|\}$，$\sigma(x)$ 表示 x 的后缀是 p 的前缀的最大长度，即 $\sigma(x) = \max_{i=0}^{|p|} i[suf(x, i) = pre(p, i)]$（中括号表示 艾佛森括号，括号内表达式为真时为 1，否则为 0）。例如，对于模板串 aba，$\sigma(c) = 0$，$\sigma(a) = 1$，$\sigma(cccaba) = 3$。

后缀函数的意义在于，对于文本串的前缀 $pre(s, i)$，找到最大 $0 \le j \le i$，使得 $suf(pre(s, i), j) = pre(p, j)$，即对于文本串的每个前缀，找到能够匹配模板串的最长后缀。继续优化算法，我们试图对于文本串的每个前缀计算它的后缀函数值，如果出现后缀函数值为 $|p|$，那么表示发现了一次模板串的出现。

下面我们来观察当从左到右扫描文本串时，后缀函数值会如何发生变化。

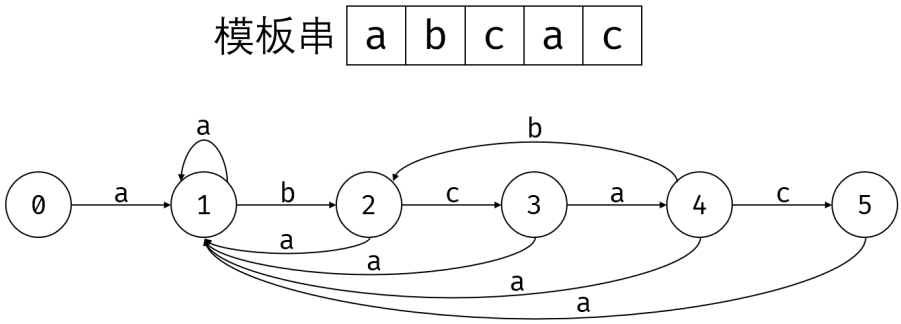


对于模板串 $abcac$，在第一步时文本串匹配好了它的前缀 $abca$，后缀函数值为 4，第二步往后扫描到了字符 b，匹配到了 ab，后缀函数的值为 2，第三步往后扫描了字符 c，匹配到了 abc，后缀函数值为 3。在每一步后缀函数变化时，根据上面结论的启发，需要使用当前的匹配信息进行优化，即当前的后缀函数值。或许有这么一个猜想，每一次的后缀函数的变化，只和前一次的后缀函数值和当前扫描到的字符有关，这个猜想是正确的，于是我们终于引入到了本节的标题 —— 字符串匹配自动机。

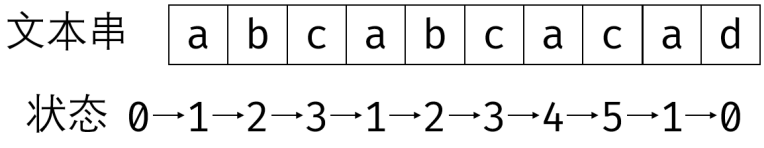
对于模板串 p，定义字符串匹配自动机状态集合是 $\{0,1,2,\ldots,|p|\}$，初始状态是 0，终止状态集合是 $\{|p|\}$，转移函数是 $\delta(x,a)=\sigma(pre(p,x)a)\ (a\in\Sigma)$。这个转移函数实际就是上面的猜想，假如在文本串 s 上匹配到了第 i 位，令 $x=\sigma(pre(s,i))$，那么在文本串中读入第 $i+1$ 位置的字符 s_{i+1} 后，串 $pre(p,i+1)$ 的后缀函数值就等于 $pre(p,x)s_{i+1}$ 的后缀函数值，而与文本串位置 $i+1-x$ 之前的字符内容无关。

假如我们已经对于模板串的每个前缀（状态）和后继的字符，都预处理出了其转移函数值，我们就可以在 $O(|s|)$ 的时间复杂度内，找到模板串在 s 中的所有出现位置。初始时，文本串是一个空串，位于起始状态 0，每次读入一个字符，在当前状态找到对应字符的转移函数，转移状态，如果位于状态 $|p|$，报告发现一次模板串的出现。如果每次查找转移函数表，进行状态转移的时间复杂度是 $O(1)$ 的，那么整体匹配的时间复杂度是 $O(|s|)$。

下图为模板串 $abcac$ 的状态转移图，其中未标出的转移边均连到 0 状态。



匹配文本串 $abcabcacad$ 的过程如下所示，因此发现了唯一的出现位置 4 (结束于 8)。



状态 0→1→2→3→1→2→3→4→5→1→0

总结，本节我们首先通过观察暴力算法匹配时，可以通过维持已匹配后缀信息的方式，减少之后匹配的比较次数。随后给出了后缀函数的定义，提出一个优化的算法，从左到右依次扫描字符串，维持文本串每个前缀的后缀函数值。利用维护的后缀信息，借助字符串匹配自动机，我们能够高效地实现向扫描的文本串下一个字符，并且得到对应后缀函数值，从而求出所有模板串的出现位置。

但是，本节只是假定你能够预处理出转移函数表，但是并未给出实际做法，这一部分内容将使用 Aho-Corasick 算法给出；另外本节还不加证明的使用了一个结论，令 $x=\sigma(pre(s,i))$，$\sigma(pre(s,i+1))=\sigma(pre(p,x)s_{i+1})$，以下给出简要证明。

> 引理 1 后缀函数不等式：$\forall x\in\Sigma^*,a\in\Sigma$，有 $\sigma(xa)\le\sigma(x)+1$。
>
> 证明：根据定义 $\sigma(x)\ge 0$，若 $\sigma(xa)=0$，不等式显然成立。若 $\sigma(xa)>0$，令 $r=\sigma(xa)$，则 $suf(xa,r)=pre(p,r)$，那么去除掉最后一个字符 a，$suf(x,r-1)=pre(p,r-1)$，根据 $\sigma(x)$ 的定义，显然 $r-1\le\sigma(x)$，即 $r\le\sigma(x)+1$。
>
> 后缀函数不等式指出，后缀函数值不会发生突然增加，每次最多 $+1$，但是可能出现突然减少。
>
> 引理 2：$\forall x\in\Sigma^*,0\le i\le|x|$，$\sigma(suf(x,i))\le\sigma(x)$。

> 证明：根据后缀函数的定义，存在 j $(i < j \leq |x|)$，满足 $suf(x, j) = pre(p, j)$，那么 $\sigma(x) > \sigma(suf(x, i))$，否则，$\sigma(x) = \sigma(suf(x, i))$。
>
> 后缀函数递归引理：$\forall x \in \Sigma^*, a \in \Sigma$，令 $q = \sigma(x)$，有 $\sigma(xa) = \sigma(pre(p, q)a)$。
>
> 证明：因为 $pre(p, q)$ 是 x 的后缀，因此 $pre(p, q)a$ 是 xa 的后缀。设 $r = \sigma(xa)$，根据后缀函数不等式，$r \leq q + 1$，所以 $suf(xa, r)$ 是 $pre(p, q)a$ 的后缀，因此 $\sigma(suf(xa, r)) \leq \sigma(pre(p, q)a)$，显然 $\sigma(suf(xa, \sigma(xa))) = \sigma(xa) = r$，$\sigma(xa) \leq \sigma(pre(p, q)a)$。由于引理 2，$\sigma(pre(p, q)a) \leq \sigma(xa)$ 。所以，$\sigma(xa) = \sigma(pre(p, q)a)$ 。
>
> 令 $x = pre(s, i)$，$a = s_{i+1}$，可知我们使用的结论正确。
>
> 上述证明参考自算法导论相关章节。

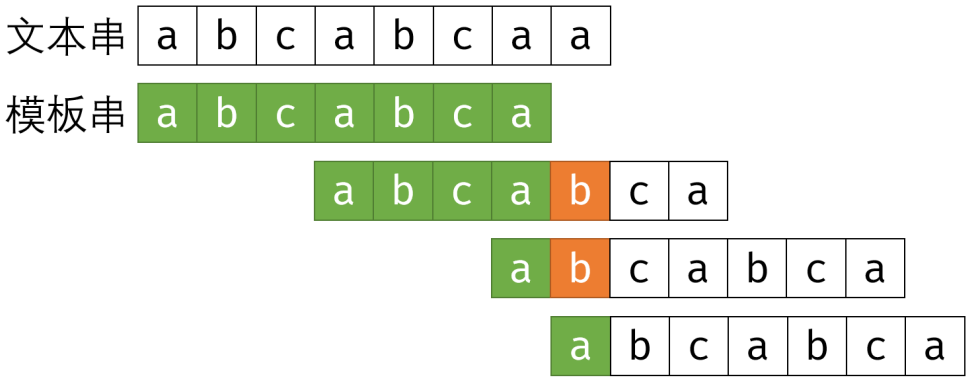
Knuth–Morris–Pratt 算法

Knuth–Morris–Pratt 算法，简称 KMP 算法，是由 James H. Morris 构思设计，Donald Knuth 受到自动机理论启发独自发现，Morris 和 Vaughan Pratt在 1970 年首次发表，并在 1977 年由 3 人共同发表这个算法。这是历史上首个线性时间的字符串匹配算法。

> 引用自 Knuth–Morris–Pratt algorithm。

在后缀函数的基础上，对于模板串 p，定义前缀函数 $\pi(x) : \{0, 1, 2, \ldots, |p|\} \to \{0, 1, 2, \ldots, |p|\}$，表示模板串长度为 x 的前缀的最长真后缀，满足其也是模板串的前缀，即 $\pi(x) = \max_{i=0}^{x-1} i[pre(p, i) = suf(pre(p, x), i)]$。其中 $\pi(0) = \pi(1) = 0$。前缀函数在国内通常也被称作 KMP 的 fail 数组。

以模板串 $abcabca$ 为例，$\pi(0) = 0$，$\pi(1) = 0$，$\pi(2) = 0$，$\pi(3) = 0$，$\pi(4) = 1$，$\pi(5) = 2$，$\pi(6) = 3$，$\pi(7) = 1$。仍然使用类似于上一节的方式，每次扫描一个字符都维护出后缀函数，假设文本串已经匹配好了 $abcabca$，现在往后扫描一个字符 a，它并没有与模板串的下一个位置匹配（模板串已经没有字符），回忆上一节的开头结论，因为需要保证已匹配部分内容的不冲突，滑动模板串的格数需要满足相应长度的前后缀相同，实际上就类似上面定义的前缀函数，将模板串向右滑动 $|p| - \pi(|p|)$ 格，再次检查对应位置是否匹配，若匹配成功，则得到当前的后缀函数值，否则重复上述的滑动。这个过程就是在每个位置原地计算出需要的转移状态，而不是预处理整个转移函数表。



这样做是没有问题，因为，假如当前的匹配长度为 i，如果向右滑动的格数不够，则根据定义，是与已匹配内容矛盾的。因此第一次滑动的格数就是 $i - \pi(i)$，即匹配长度变成 $\pi(i)$。再次滑动就是长度变成 $\pi(\pi(i)), \pi(\pi(\pi(i))), \ldots$，即枚举 i 结尾的所有长度，满足相应长度的前后缀相同，不会出现其他长度满足这个条件。因为，不妨假设存在 $\pi(\pi(i)) < j < \pi(i)$，j 满足 $pre(p, j) = suf(p, j)$，那么这就与 $\pi(\pi(i))$ 的值矛盾。

> 更详细和完整的证明可参考算法导论相关章节。

于是，我们得到一个使用前缀函数，而非使用自动机的维护后缀函数值的算法，以下先给出它的伪代码。

Input. Template string p, text string s.
Output. All the occurence of p in t.
Method.

```
1  π ← get prefix function of p
2  curState ← 0
3  for i ← 1 to |s|
4    while curState ≠ 0 and p_{curState+1} ≠ s_i
5      curState = π(curState)
6    if p_{curState+1} = s_i then curState ← curState + 1
7    if curState = |p| then
8      find an occurence i − |p| + 1
9      curState = π(curState)
```

上述伪代码的 4 到 6 行即为求出新的后缀函数值，转移不再是使用直接查表，而是一种近乎暴力的方式。但是实际上，考虑 curState 的变化，它只会增加 $O(|s|)$ 次，而在第 4 行和 5 行的循环内，curState 都会至少减少 1，因此总的时间复杂度是 $O(|s|)$。

最后的问题就是，我们仍然不能求出前缀函数，观察上述的伪代码，实际上每次更新状态只会用到小于 i 的前缀函数值，只需要使用模板串 p 去匹配自身，还是可以维护出每个位置的状态，即前缀函数值。时间复杂度是 $O(|p|)$，空间复杂度 $O(|p|)$。显然 KMP 算法的时间和空间复杂度是与 $|\Sigma|$ 无关的。

总结，我们使用 KMP 算法解决问题一，时间复杂度 $O(|s|+|p|)$，空间复杂度 $O(|p|)$。KMP 算法是字符串匹配自动机的一个精简高效的实现，它使用时空代价为 $O(|p|)$ 前缀函数，取代字符串匹配自动机 $O(|p||\Sigma|)$ 的转移函数表，减少了复杂度中的 Σ 因子。

字符串的整周期

定义字符串 s 的 k 次幂串为 k 个 s 连接起来的字符串，记为 $s^k = ss\dots s$（k 个 s）。定义字符串 s 有整周期 p，满足 p 整除 $|s|$，$pre(s,p)^{\frac{|s|}{p}} = s$。

对于字符串 s，若 $0 \le r < |s|$，$pre(s,r) = suf(s,r)$，就称 $pre(s,r)$ 是 s 的 `border`。若 $0 < p \le |s|$，$\forall 1 \le i \le |s|-p, s_i = s_{i+p}$，就称 p 是 s 的**周期**。注意和整周期的区别，周期允许最后“缺少一块”，但是如果周期 p 整除 $|s|$，那么周期 p 也是 s 的整周期。

周期和 `border` 两者之间存在等价关系，有引理：若 t 是 s 的 `border`，当且仅当 $|s|-|t|$ 是 s 的周期。

证明：

- 若 t 是 s 的 `border`，那么 $pre(s,|t|) = suf(s,|t|)$，因此 $\forall 1 \le i \le |t|, s_i = s_{|s|-|t|+i}$，所以 $|s|-|t|$ 就是 s 的周期。
- 若 $|s|-|t|$ 为 s 周期，则 $\forall 1 \le i \le |s|-(|s|-|t|) = |t|, s_i = s_{|s|-|t|+i}$，因此 $pre(s,|t|) = suf(s,|t|)$，所以 t 是 s 的 `border`。

因此，注意到 `border` 的定义实际上就是 KMP 算法中的前缀函数，那么对于字符串 s，求出它的前缀函数值，那么就可以求出它的所有 `border`，等价于求出它的所有周期。关于如何求出所有 `border`，回忆 KMP 算法中维护状态的方式，实际上就是 $\pi(|s|), \pi(\pi(|s|)), \dots$。

如果所求的是整周期，那么只需要判断周期是否整除串长即可。

练习题：HDu1358 Period。

在 2019 年 CCPC 秦皇岛赛区现场赛中也出现了类似的练习题：HDu6740 MUV LUV EXTRA。

统计前缀的出现次数

给定字符串 s，你现在对于每个 i $(1 \le i \le |s|)$，求出 $pre(s,i)$ 在 s 中的出现次数。

其中 $1 \le |s| \le 10^5$。

我们可以枚举每个前缀，统计这个前缀有多少个后缀是原串的前缀。回忆 KMP 算法中维护状态的方式，假设当前枚举到长度为 i 的前缀，就是出现了长度为 $i, \pi(i), \pi(\pi(i)), \ldots$ 的前缀，这些前缀的出现次数都会 $+1$。

但是如果我们求前缀函数中，暴力跳到底，KMP 时间复杂度的均摊分析就失效了，算法的时间复杂度将退化成 $O(|s|^2)$。

注意到事实 $\forall 1 \le i \le |s|, \pi(i) < i$，那么将 $\pi(i)$ 作为结点 i 的父亲，那么 0 到 $|s|$ 这些结点构成了一棵以 0 为根的树。如果出现了状态 i，就等价于在结点 i 到根的路径上，所有点的出现次数 $+1$。

但是并不需要真的全部 $+1$，只需要在每个点打好标记，最后遍历一次这棵树，对于每个结点统计子树内的权值和即可。在这个问题中，我们甚至不需要显示建出这棵树，因为实际上已经有了这整棵树的拓扑序，即 $0, 1, 2, 3, \ldots, |s|$， 只需要逆序遍历拓扑序就可以实现统计子树的权值和，也就是每个前缀的出现次数。

这个做法也非常容易扩展到求模板串每个前缀在文本串的出现次数，也就是下面的例题留给读者练习。

练习题：HDu1358 A Secret。

例题 1

给定一个字符串 s，对它的每个前缀，求出满足既是这个串的前缀也是其后缀，并且前后缀不重叠的最大长度。

其中 $1 \le |s| \le 10^6$。

来源：「NOI2014」动物园

例题 2

给定一个带通配符的字符串 s，两个字符串 a 和 b，你现在需要将 s 中的通配符都使用小写字母替代，最大化 a 串在 s 串中的出现次数减去 b 串在 s 串中的出现次数。

其中 $1 \le |s| \le 1000, 1 \le |a|, |b| \le 50$。

来源：Codeforces Round #558 (Div. 2) D. Mysterious Code。

例题 3

强制在线，有一个空串和一个空数组，n 次操作，每次向后添加一个字母和一个值，定义一个子串的价值为数组对应区间的最小值，询问该串的所有子串的价值之和，子串需要满足其等于相应长度前缀的条件。

其中 $1 \le n \le 600000$。

来源：Codeforces Round #612 (Div. 1) E. Fedya the Potter Strikes Back。

例题 4

定义 `half border` 为长度不超过一半的 `border`。

给定一个长度为 n 的字符串 s，q 次询问，每次独立地删除一个子串 $[l_i, r_i]$，左右连起来，询问这个串的 `half border` 长度和。

其中 $1 \le n, q \le 5 \cdot 10^5$。

来源：Comet OJ - Contest #12 F. Substring Query。

Z 算法

Z 算法，也就是国内俗称的扩展 KMP 算法。

定义 Z 函数表示，第 i 个位置开头的后缀和原串的最长公共前缀。

对于字符串 s 求出它的 Z 函数的时间复杂度是 $O(|s|)$，算法具体内容可以参考以下资料。

- Z 函数（扩展 KMP）；
- Z-function and its calculation。

例题：Codeforces Round #622 (Div. 2) E. Concatenation with intersection。

Shift-And 算法

Shift-And 算法，也被称为 Shift-Or 算法，通过下面这道例题介绍，也可参考 这篇博客的可视化讲解。

字符集为 0 到 9 的数字，现在给定一个长度为 n 的形如 $(0|1|2)(3|4)(5|6)\dots$ 的正则表达式，求输入串 s 有多少个子串可以匹配这个正则表达式。

其中 $1 \le n \le 1000$，$1 \le |s| \le 5 \cdot 10^6$。

来源：HDu5972 Regular Number。

令 $dp(i,j)$ 表示考虑到 s 串的 i 个位置，是否能够匹配到正则表达式的第 j 个位置，转移就是：

$$dp(i,j) = dp(i-1,j-1) \wedge [s_i \text{ matches } RegExp_j]$$

如果我们把 $dp(i)$ 表示表示成长度为 n 的二进制状态，那么这个转移过程可以变成前一个位置的匹配状态左移一位，和 s_i 能够匹配的位置表示的二进制状态进行二进制与操作，就能批量完成转移。

时间复杂度：$O(|s|\frac{n}{64})$。Shift-And 算法实际上就是利用位运算（`std::bitset`）的暴力算法。

Rabin-Karp 算法

Rabin-Karp 算法本质是利用 Hash 的暴力匹配算法。Hash 的核心思想在于将输入映射到一个值域比较小的范围，从而进行快速地比较。

在算法竞赛中，常用的字符串哈希方式是将字符串表示成一个 $base$ 进制下对某个模数 p 取模的一个整数，定义 $f : s \in \Sigma^* \to \mathbb{N}$。

$$f(s_0 s_1 \dots s_{|s|}) = (s_0 base^0 + s_1 base^1 + \dots + s_{|s|} base^{|s|}) \bmod p$$

建议：不要使用自然溢出，即对 $2^{32}-1$ 取模，Hack 方式；模数可以选择一些大质数，并且使用多个模数多重校验提高准确率。不建议在 Codeforces 上的一些简单字符串问题使用哈希。

使用以上哈希方式，如果我们预处理每个前缀的哈希值，我们将容易计算出每个子串的哈希值，$f(s_{l\dots r}) = \frac{f(s_{1\dots r}) - f(s_{1\dots l-1})}{base^{l-1}}$。

Rabin-Karp 算法就是使用文本串中每个位置的哈希值，来对是否出现模板串进行初步校验，如果哈希值相同进行暴力比较。时间复杂度：$O(|s||p|)$，空间复杂度 $O(1)$。

Aho–Corasick 算法

问题 2

给定 n 个模板串 p_i $(1 \le i \le n)$ 和 1 个文本串 s，对于每个模板串求出它在文本串中的出现次数。

其中 $1 \le \sum_{i=1}^{n} |p_i|, |s| \le 10^5$。

来源：【模板】AC自动机（二次加强版）。

暴力

我们已经学会了 KMP 算法，那我们来试试吧！

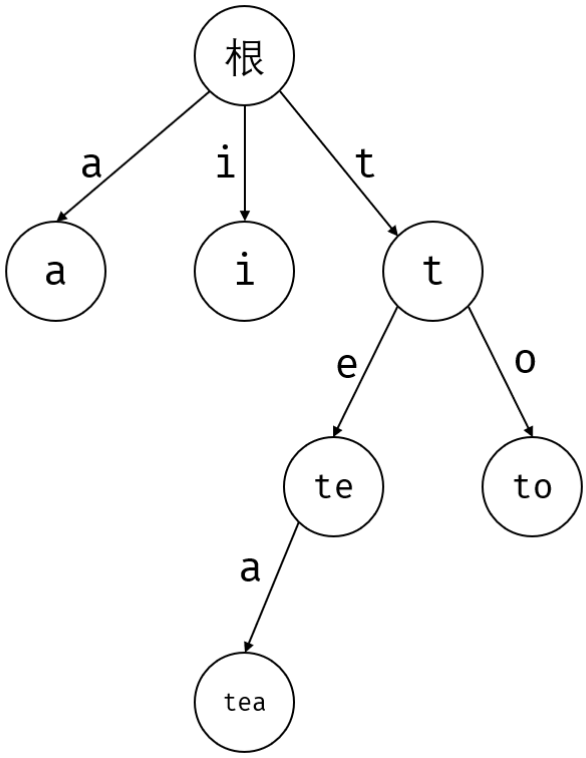
对每个模板串，构建前缀函数，用前缀函数在文本串上进行匹配。

分析时间复杂度，每个模板串构造前缀函数的时间复杂度是 $O(\sum_{i=1}^{n}|p_i|)$，使用每个前缀函数匹配文本串的时间复杂度是 $O(n|s_i|)$。如果模板串数量很多，文本串很长，这个时间复杂度显然是难以接受的。

Trie

上面的暴力算法是对于每个模板串单独考虑，每次都扫描了一遍文本串，因此我们希望把所有模板串合并到一起考虑，只扫描一遍文本串。

Trie，也被称为字典树或前缀树，在 1959 年由 René de la Briandais 首次提出。字典树的结构如下图所示。



字典树是一棵有根树，**每条边**上有一个字符，每个结点代表一个字符串，它是从根结点到这个结点的路径上每条边的依次连接起来形成的字符串。上图中，每个结点代表的字符串被标在结点中，注意根结点代表空串。

构造方法，从左至右扫描要插入的字符串，如果结点没有被创建，那么创建一个新的结点，然后继续往下构造字典树。参考代码如下：

```cpp
std::map<int, char> ch[maxn];
int insert(char* s) {
  int u = 0;
  for (int i = 0; s[i]; i++) {
    if (!ch[u][s[i]]) {
      ch[u][s[i]] = ++sz;
      ch[sz].clear();
    }
    u = ch[u][s[i]];
  }
  return u;
}
```

字典树也是解决一些与异或等与二进制位有关问题的重要工具，例如询问一个数与一个集合内某个数异或的最大值或最小值问题。

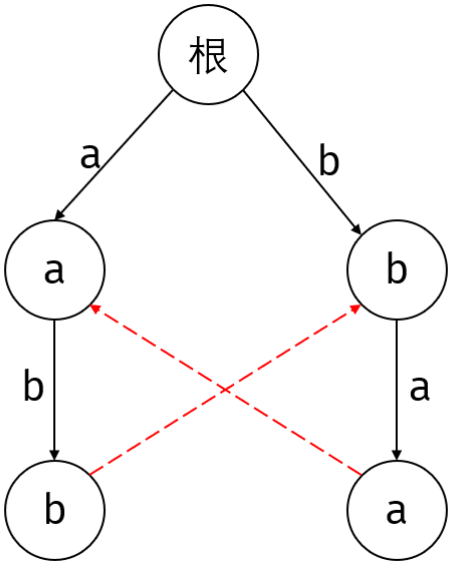
Aho–Corasick 算法

Aho–Corasick 算法，也就是俗称的 AC 自动机，他是由 Alfred V. Aho 和 Margaret J. Corasick 两人在 1975 年发明。

AC 自动机也是一个字符串匹配自动机，它的状态是所有模板串构造的字典树中所有结点，起始结点是字典树的根结点，终止结点是所有模板串的在字典树最后一个结点。在上一章的字符串匹配自动机中，转移函数为往后添加一个字符的后缀函数值的对应状态，而单模板串的问题本质就是只有一根长链的字典树，我们将后缀函数的定义扩展到字典树上，也可以类似地为 AC 自动机定义出转移函数。

定义后缀函数 $\sigma : \Sigma^* \to \text{Trie 结点集合}$，设字符串 s，则 $\sigma(s)$ 表示 s 的最长后缀对应的 AC 自动机结点，这个结点必须是存在的。显然这个定义是完备的，因为字典树的根结点是空串，而空串必定是一个字符串的后缀。AC 自动机的转移函数 δ 就是 $\delta(x, a) = \sigma(T(x)a)$（$T(x)$ 表示结点 x 在字典树上代表的串）。

类似于 KMP 算法，我们还能类似地定义前缀函数，我们一般称其为 $fail$ 指针或者后缀链接。定义 AC 自动机上一个结点的后缀链接，就是这个结点代表串的最长真后缀对应的 AC 自动机结点，这个结点必须是存在的。下图为 ab 和 ba 两个串的 AC 自动机，其中红色虚线为 fail 指针。



到现在为止，所有的定义从字符串匹配自动机和 KMP 算法自然扩展而来，AC 自动机的后缀链接构造算法也可以简单修改计算前缀函数的算法得到。在前缀函数时，暴力上跳上一个状态的前缀函数，直到找到匹配位置为止，然后转移到下一个状态。这样做必须保证之前每个位置的前缀函数值都已经计算完成，而 AC 自动机是一个树形结构，并非简单的线性序列，如果采用 dfs 字典树构造的方式，可能会存

在当前状态依赖未构造部分的情况，但是如果采用 bfs 的方式，一层一层地计算后缀链接，就能保证正确性。伪代码如下所示。

Input. Trie of $\{p_1, p_2, \ldots, p_n\}$.
Output. Suffix link of the Trie.
Method.

```
1  Queue ← empty
2  fail ← (null, null, ...)
3  for each a ∈ Σ then
4    if root has an edge a then
5      push δ(root, a) into Queue
6      fail(δ(root, a)) ← root
7  while Queue is not empty then
8    u ← pop the front node of Queue
9    for each a ∈ Σ then
10     if u has an edge a then
11       state ← fail(u)
12       while state does not have an edge a then
13         state = fail(state)
14       fail(δ(u, a)) ← δ(state, a)
15       push δ(u, a) into Queue
```

上述算法类比 KMP 求前缀函数的证明，时间复杂度是 $O(\sum_{i=1}^{n}|p_i|)$。因为状态转移时，串长 $+1$，使用后缀链接转移时长度至少 -1，一个结点到根的路径上使用后缀链接上跳的次数不会超过结点的深度，因此整体上不超过**所有叶子结点的深度和**。同样也能证明，使用 AC 自动机去匹配文本串 s 的时间复杂度是 $O(|s|)$。

在字符串匹配自动机一章，我们留下了一个悬念，高效构造自动机转移函数的算法。国内一般将 Aho-Corasick 算法显示构造出真正的转移函数得到自动机称为 Trie 图。

> 参考资料 Trie图的构建、活用与改进。

构造后缀链接时，对于结点 u 和字符 a，如果字典树上 u 有一条往下字符为 a 的边，那么 $\delta(u,a)$ 直接使用字典树上的转移边即可，而这个 $\delta(u,a)$ 的 fail 指针指向 u 的 fail 指针指向的结点的 a 转移函数值，即 $\text{fail}(\delta(u,a)) = \delta(\text{fail}(u),a)$。因为 $\text{fail}(u)$ 的深度一定小于 u，这部分因为 bfs 已经计算完毕，所以直接使用即可。如果字典树上 u 没有 a 转移边，那么 $\delta(u,a) = \delta(\text{fail}(u),a)$。这个过程本质和原本的暴力没有任何区别，可以简单理解为一种记忆化或者动态规划进行递推，列出式子如下：

$$\delta(u,a) = \begin{cases} \delta(u,a), & \text{if } u \text{ has an edge } a \\ \delta(\text{fail}(u),a), & \text{if } u \text{ does not have an edge } a \end{cases}$$

$$\text{fail}(\delta(u,a)) = \delta(\text{fail}(u),a), \text{if } u \text{ has an edge } a$$

上述构造 Trie图的算法时间复杂度是 $O(\sum_{i=1}^{n}|p_i||\Sigma|)$，空间复杂度是 $O(\sum_{i=1}^{n}|p_i||\Sigma|)$。

最后，回到原问题，如何统计出每个模板串的出现次数？联系 KMP 算法的应用时介绍的统计前缀出现次数的方式，每次匹配到一个结点，等价于 fail 树上这个点到根的路径上所有点权值 $+1$，扫描文本串时打标记，最后建出fail 树 dfs 或者拓扑排序，求出每个点在 fail 树上的子树权值和即可。时间复杂度 $O(|s|+\sum_{i=1}^{n}|p_i|)$。

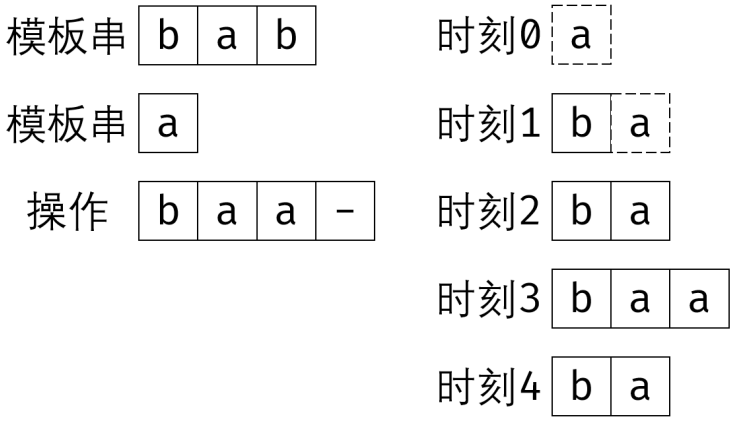
Trie 图

例题 1

给定 n 个模板串 p_i 和一个带退格的操作序列 s，对于操作序列的每个前缀回答至少需要添加几个字符，使得添加后的后缀是某个模板串。

其中 $1 \leq n \leq 4, 1 \leq |p_i|, |s| \leq 10^5$。

来源：牛客网暑期ACM多校训练营（第九场）F. Typing practice。



例题 2

字符集为 1 到 10^5 中的所有正整数，输入一棵大小为 n 的字典树，要求你构造出每个点的 fail 指针。

其中 $1 \leq n \leq 10^5$。

来源：ICPC Camp 2016 Day 1 A. Aho-Corasick Automaton。

例题 3

给定 n 个模板串 p_i，求是否存在一个无限长的字符串，不包含任何一个模板串。

其中 $1 \leq \sum_{i=1}^{n} |p_i| \leq 3 \times 10^4$。

来源：「POI 2000」病毒。

例题 4

给定 n 个模板串 p_i，求有多少个长度为为 k 的字符串，不包含任何一个模板串。

其中 $1 \leq \sum_{i=1}^{n} |p_i| \leq 100, 1 \leq k \leq 2 \times 10^9$。

来源：「HNOI2008」GT考试，PKu2778 DNA Sequence。

AC 自动机和动态规划

在下一章后缀自动机中也有一道类似的习题。

例题 1

给定 n 个模板串 p_i 和一个文本串 s，你需要用这 n 个模板串去覆盖文本串，覆盖必须保证匹配，一个位置可能被多个模板串覆盖，一个模板串可以被多次使用，求最少需要多少个模板串将文本串完整覆盖。

其中 $1 \leq |s|, \sum_{i=1}^{n} |p_i| \leq 3 \times 10^5$。

来源：CERC 2018. A. The ABCD Murderer。

例题 2

给定 n 个模板串 p_i 和一个文本串 s，每个模板串有花费 $cost_i$，要求将母串划分为数段，每段均为一个模板串，代价是用到的所有串的花费之和，求拼出文本串的最小代价。

其中 $1 \leq |s|, \sum_{i=1}^{n} |p_i| \leq 5 \times 10^5, 1 \leq cost_i \leq 10^9$。

来源：2020 CCPC Wannafly Winter Camp Day2 K. 破忒头的匿名信。

例题 1

初始有一个空串，共有 n 次操作，每次会有 3 种操作：

1. 添加一个字符；
2. 删除最后一个字符；
3. 打印当前的字符串。

有 q 次询问，每次询问第 x 次打印的字符串在第 y 次的字符串中出现多少次。

其中 $1 \leq n, q \leq 10^5$。

来源：「NOI2011」阿狸的打字机。

例题 2

给定 n 个字符串 s_i，你现在有空的字符串集合 T，共有 q 次操作，每次会有 3 种操作：

1. 向集合 T 内添加一个字符串 p_i；
2. 询问 T 中有多少个字符串包含 s_x。

其中 $1 \leq n, q \leq 10^5, 1 \leq \sum_{i=1}^{n} |s_i|, \sum_{i=1}^{q} |p_i| \leq 2 \times 10^6$。

来源：「Coci2015」Divljak。

例题 3

给定一棵大小为 n 字典树，根到每个点的路径上字符组成的串的反串对应每个人的名字，有 q 次询问，每次输入一个串 s_i，询问这个串是多少个人名字的前缀。

其中 $1 \leq n, q, \sum_{i=1}^{n} |s_i| \leq 10^6$

来源：ICPC 2019, World Finals G. First of Her Name，Educational Codeforces Round 71 (Rated for Div. 2) G. Indie Album。

例题 4

给一个长度为 n 的带有通配符的字符串 s 和 m 个模板串 p_i，每个模板串有权值 v_i，你现在需要将这个字符串的所有通配符替换为具体字符，最大化权值。定义 s 的权值 $Magic$ 初始为 1，若出现模板串 i 则权值乘 v_i，假设最终出现 c 个模板串（同一模板串多次出现算多次），最终权值为 $\sqrt[c]{Magic}$。

其中 $1 \leq n, m, \sum_{i=1}^{n} |p_i| \leq 1501, 1 \leq v_i \leq 10^9$。

来源：「BJOI2019」奥术神杖。

例题 5

给定一个 $R \times C$ 的二维方格，有 Q 次询问，每次输入一个串 s_i，求二维方格中有多少个这个串。

定义二维方格中的一个串，它是由一段从左到右横着的串，后面接着一段从上到下的串，两段拼在一起。

其中 $1 \leq R, C \leq 500, 1 \leq Q, \sum_{i=1}^{Q} |s_i| \leq 2 \times 10^5$。

来源：2019-2020 ICPC, Asia Jakarta Regional Contest D. Find String in a Grid。

Trie 图上的随机游走

问题描述

给定 n 个长度均为 m 的字符串，现在你开始投骰子，有 P_c 的概率投出字符 c，求第一次出现的串是第 i 个的概率。

一般做法

对这个字符串集合建出 Trie 图后，问题等价于给定一些终点的无向图随机游走问题。

令结点数为 s，建出概率转移矩阵后，实际上只有 $s-1$ 条有效的方程，因为无法转移到根结点。但是，随机游走必定会在某个时刻终结，也就是走到所有终止结点的概率和为 1。添加这么一条方程后，高斯消元，即可解出在每个终止结点结束的概率。

时间复杂度：$O(nm|\Sigma|+n^3m^3)$。

例题：「Jsoi2009」有趣的游戏。

加强做法

令概率生成函数 $[x^k]G(x)$ 表示在 k 时刻尚未停止的概率，$[x^k]F_i(x)$ 表示在 k 时刻末尾是第 i 个串的概率。

我们要求的就是 $F_i(1),F_i(2),\ldots,F_i(n)$。

考虑在每个尚未停止的时刻，往后走一步的情况，即 $xG(x)$，再加上初始时的概率 1，这时要么在此处迎来某个串的结束，要么尚未停止，有下式。

$$1+xG(x)=G(x)+\sum_{i=1}^{n}F_i(x)$$

记 $P(S)$ 表示一个串的出现概率，即 $P(S)=\prod_{i=1}^{|S|}P_{S_i}$

对于每个串 i，在一个未终止时刻，往后加第 i 个串，即 $G(x)P(s_i)x^m$。这时情况很多，因为加这个串的时候可能中途出现已经终止的情况，设第 j 个串在新加串的 k 位置终止，因此 $s_i[1\ldots k]=s_j[m-k+1\ldots m]$，即第 i 个串的前缀等于长度为 k 的第 j 个串的后缀，此时的概率生成函数就是 $F_j(x)P(s_i[k+1\ldots m])x^{m-k}$。

我们枚举每个串 j 和它的前缀长度 k，有下式：

$$G(x)P(s_i)x^m=\sum_{j=1}^{n}\sum_{k=1}^{m}[s_i[1\ldots k]=s_j[m-k+1\ldots m]]F_j(x)P(s_i[k+1\ldots m])x^{m-k}$$

结合我们要求的，带入 $x=1$，可以得到 n 个方程，但是 $G(1)$ 也是未知量，因此有 $n+1$ 个未知量，联立 $\sum_{i=1}F_i(1)=1$，即可得到 $n+1$ 个方程，高斯消元。注意这里我们的变量数是 $n+1$，不是一般做法的最大 nm。

时间复杂度：$O(n^2m+n^3)$。

例题：「SDOI2017」硬币游戏。

> 参考资料 2018 年集训队论文 浅谈生成函数在掷骰子问题上的应用 杨懋龙。

后缀树

问题 3

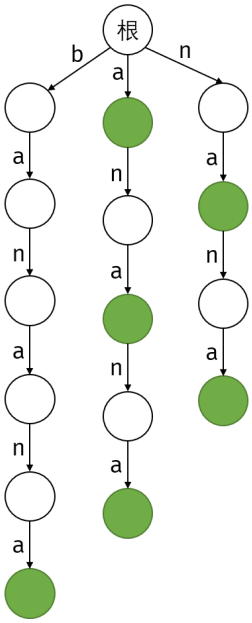
给定 1 个模板串 p，有 q 次询问，每次询问一个文本串 s_i 在模板串中的出现次数。

其中 $1\le|p|,\sum_{i=1}^{q}|s_i|\le10^5$。

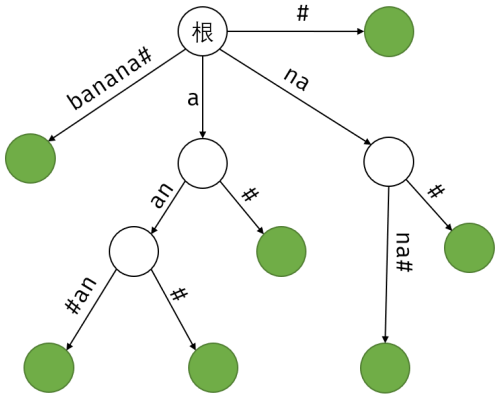
后缀结构

后缀树，后缀自动机，后缀数组三者的本质几乎是相同的，只是呈现的形式，构造的方法，如何使用有着一些差异而已，我们需要根据需求选择恰当的后缀结构才能高效地解决实际问题。不过在算法竞赛中，后缀自动机由于其本身蕴含着大量的信息，同时兼顾代码实现非常简单的特征，因而被广泛应用，这里必须感谢陈立杰老师对后缀自动机在算法竞赛中的推广做出的巨大贡献。但是，后缀自动机本身理解难度较高，因此这里我将着重介绍后缀树的结构，然后将其与后缀自动机的一些性质和特征进行联系。

在上一章中已经介绍过了字典树，后缀字典树就是将一个字符串的所有后缀建出的字典树。$banana$ 的后缀字典树如下图所示，其中标绿的结点是终止结点。



显然，后缀字典树的结点数很有可能退化成 n^2 个结点。定义压缩字典树，是将一棵字典树上的所有二度结点（只有父亲和一个儿子的结点）进行合并后得到的字典树，而后缀树就是压缩的后缀字典树。注意到，在上面的例子中，有一些后缀不是叶子结点，这将导致在后缀树中其被压缩到某条边中，即隐式后缀树。解决方法十分简单，将原来的模板串最后加一个字符集外的字符 #，这样所有后缀都是是一个叶子结点，我们称这样的后缀树为显式后缀树。$banana$ 的显式后缀树如下图所示。



有了后缀树的定义，就能推出一些显然的结论和应用。

对于字符串 s，后缀树的结点数至多有 $2|s|-1$ 个（当后缀树是一棵二叉树时，达到结点数量的最大值）。

s 的任何一个子串，都是后缀字典树上的某一个结点（子串就是某个后缀的前缀）。如果我们将每条边的信息记录在深度较大的那个点上，每个点记录和父亲连接的边上的字符在原串上的对应区间，那么 s 的一个子串就能表示为（后缀树结点，使用了哪条边，在边上走了多少个字符）这样的三元组状态。从自动机的角度上看，后缀树上的所有状态都是原串的一个子串，终止状态是原串的某一个后缀。

定义 $len(u)$ 表示对于结点 u 代表的串的长度，$fa(u)$ 表示 u 在后缀树的父亲。那么对于每个结点非根结点，就代表长度在 $(len(fa(u)), len(u)]$ 的一系列串，并且这些串在原串中出现的左端点集合都相同，这个集合就是 u 的子树内所有叶结点（后缀）出现位置的并集。考虑子树内某个叶子出现，那么其到根路径上所有结点都会在这个位置出现，因为叶子是原串的以某个位置 i 开头的后缀，而路径上所有结点都是它的前缀。而且对于结点 u 的父边长度在 $(len(fa(u)), len(u)]$ 范围内的串，由于没有产生分叉，因此这部分中的每个串出现位置集合都相同。

观察后缀树的结构容易发现，s 的两个后缀 i, j 的最长公共前缀（Longest Common Prefix，简称 LCP），就是后缀 i 和后缀 j 对应后缀树上的两个结点的最近公共祖先对应的串。

后缀树的构造可以使用 1995 年 Esko Ukkonen 发表的算法。它的时间复杂度是 $O(n)$，如果字符集大小较大可能需要使用额外的平衡树等结构维护出边，时间复杂度是 $O(n \log |\Sigma|)$。算法过程相对比较复杂，这里不详细进行介绍，感兴趣的读者可以阅读 中文讲解。Ukkonen 算法的主要思想在于，不是一个个地向字典树插入后缀，而是对已插入的叶子结点进行扩展，并且维护尚未插入的隐式后缀。算法过程中也使用到了后缀链接，这里后缀链接与 Aho–Corasick 算法中的后缀链接几乎一致。Ukkonen 算法相对于后文的构造后缀自动机的 Blumer 算法的实现细节较复杂，但是更加灵活。从后缀树的角度来说，Ukkonen 算法是在线地从左至右构造后缀树，而 Blumer 算法则是离线地从右至左构造后缀树。

后缀数组是后缀树的一种精简表示，如果你构造出了原串的后缀树，那么你 dfs 这棵树时，每个结点都按照出边的字典序进行递归，那么你得到叶子结点的遍历顺序就是原串的后缀数组。后缀数组的经典应用是求出 Height 数组，其定义是后缀数组中相邻的两个后缀的最长公共前缀，而回忆后缀树上的结论，Height 数组本质就是那两个后缀的最近公共祖先的串长。构造原串的后缀数组的常用算法是 Manber & Myers 于 1990 年发表的倍增算法，时间复杂度是 $O(n \log n)$，你也可以使用基于诱导排序的 SA-IS 算法，在线性时间内构造后缀数组。如果你使用线性时间的算法构造了后缀数组和 Height 数组，你可以基于它们构造出一棵虚树，即后缀树。

后缀自动机是一个有向无环单词图，它可以接受原串的所有后缀，每个状态表示原串的一系列子串。我们可以使用 Blumer 算法在线地构造出原串的后缀自动机，具体内容可以参考 后缀自动机 (SAM) - OI Wiki。

后缀自动机本身含有两个结构：第一个是用来匹配的自动机；第二个是 parent 树，原串的 parent 树是原串的反向前缀树。用一个性质来理解第二个结构就是，后缀自动机上两个结点在 parent 树上的最近公共祖先是这个两个串最长公共后缀。因此，如果你反向扫描原串，就能构造出原串的后缀树。因此我们可以把后缀自动机的状态去类比后缀树上的结点，后缀自动机上的每个状态都是一系列结束位置 $endpos$ 集合相同的串，反过来看就是后缀树上一系列开始位置集合相同的串，而你从这个后缀树角度看的话一些 $endpos$ 上的结论将会变得十分直观和显然。

后缀自动机

关于后缀数组的例题可以前往 后缀数组 (SA) 进行学习，在此列出以下后缀自动机每种应用的模板题。

子串出现次数

- 【模板】后缀自动机

本质不同子串个数

- SDOI2016 生成魔咒

字典序第 k 大子串

- TJOI2015 弦论

最长公共子串

- SPOJ Longest Common Substring
- SPOJ Longest Common Substring II

线段树合并

- Codeforces 1037H Security
- Codeforces 666E Forensic Examination
- 洛谷5161 WD与数列

广义后缀自动机

广义后缀自动机实际上是单串后缀自动机扩展为字典树上的后缀自动机，与 KMP 算法和 Aho–Corasick 算法的关系类似。因为 Blumer 算法本身就是一个增量算法，实际上你只需要指定好最后访问的结点就可以实现字典树上的后缀自动机构造。但是这里需要注意两个细节，如果你直接重置后缀自动机的最后访问结点，可能会产生一些多余的空状态，虽然在一些问题中这可能不会影响整个算法的正确性，但是还请务必注意这里可能存在问题。第二点，如果你采用 dfs 字典树的方式构造广义后缀自动机，时间复杂度将是 $O(\text{字典树叶子深度和})$，如果你采用离线字典树，使用 bfs 的方式构造广义后缀自动机，时间复杂度是 $O(n)$（n 表示字典树结点数）。

> 参考资料 2015 年国家队论文 后缀自动机在字典树上的扩展 刘研绎。

模板题：ICPC 2019, World Finals G. First of Her Name，2019 ICPC Asia Xuzhou Regional Contest L. Loli, Yen-Jen, and a cool problem。

例题

例题 1

给定一个文本串 s 和 n 个模板串 p_i，求文本串有多少个本质不同子串不包含任何一个 p_i 作为其子串。

其中 $1 \leq |s|, \sum_{i=1}^{n} |p_i| \leq 10^5$。

来源：HDu4416 Good Article Good sentence。

例题 2

给定一个字符串 s，有 q 询问，每次询问一个字符串 t_i，求这个字符串最少可以被划分为多少段，使得每段都是 s 的子串。

其中 $1 \leq |s|, \sum_{i=1}^{q} |t_i| \leq 2 \times 10^5$。

来源：2019-2020 ACM-ICPC Latin American Regional Programming Contest G. Gluing Pictures。

例题 3

字符集是 0 到 9 的数字，给定 n 个字符串 s_i，求所有本质不同子串，作为数字的和，答案对 2012 取模。

其中 $1 \leq \sum_{i=1}^{n} |s_i| \leq 10^5$。

来源：HDu4436 str2int。

例题 4

给定字符串 s，你现在需要从一个空串开始创造出字符串 s，有两种操作，花 p 的代价添加一个字符或者花 q 的代价从当前的字符串中挑一个子串加到末尾，求最小花费。

其中 $1 \leq |s| \leq 2 \times 10^5, 1 \leq p, q \leq 10^9$。

来源：2019 Multi-University Training Contest 1 Typewriter。

例题 5

字符集为 $\{0,1\}$，给定 n 个模板串 p_i，定义一个字符串是熟悉的，当且仅当长度不小于 L，且是某个模板串的子串，如果能够将文本串 s 分割为若干段，使得熟悉的长度超过原串长度的 90%，求满足这样条件的 L 的最大值。

其中 $1 \leq \sum_{i=1}^{n} |p_i|, |s| \leq 10^6$。

来源：「CTSC2012」熟悉的文章。

例题 6

给定一个字符串 s，有 q 次询问，每次询问串 p_i，求 s 中有多少个子串与 p_i 循环同构，即可以通过循环左右移动的方式变成同一个串。

其中 $1 \leq |s| \leq 10^5, 1 \leq \sum_{i=1}^{n} |p_i| \leq 10^6$。

来源：Codeforces Round #146 (Div. 1) C. Cyclical Quest。

例题 7

给定一个字符串 s，有 q 次询问，询问输入串 t_i 有多少个本质不同子串，不是原串 $s[l \ldots r]$ 的子串。

其中 $1 \leq |s|, \sum_{i=1}^{q} |t_i| \leq 5 \times 10^5$。

来源：「NOI2018」你的名字。

例题 8

字符集为 $\{0,1\}$，给定一个字符串 s，有 q 次询问，询问长度范围在 $[l,r]$ 内的两个前缀的最长公共后缀。

其中 $1 \leq |s|, q \leq 10^5$。

来源：「雅礼集训 2017 Day7」事情的相似度。

回文串

Manacher 算法

Manacher 算法通过添加字符的方式，将奇偶长度的回文串统一处理，以求出每个位置的最长回文半径，具体算法请参考 Manacher - OI Wiki。

回文树

回文树 (EER Tree / Palindromic，也被称为回文自动机) 是一种可以存储一个串中所有回文子串的高效数据结构。它是一个新颖的字符串数据结构，最初是由 Mikhail Rubinchik 和 Arseny M. Shur 在 2015 年发表。它的灵感来源于后缀树，使用回文树可以简单高效地解决一系列涉及回文串的问题。具体算法请参考 回文树 - OI Wiki，更多应用和例题也可以参考 2017 年国家集训队论文 回文树及其应用 翁文涛。

最小回文划分

> 给定一个字符串 $s(1 \leq |s| \leq 10^5)$，求最小的 k，使得存在 $s_1, s_2, \ldots, s_k$，满足 $s_i(1 \leq i \leq k)$ 均为回文串，且 $s_1, s_2, \ldots, s_k$ 依次连接后得到的字符串等于 s。

考虑动态规划，记 $dp[i]$ 表示 s 长度为 i 的前缀的最小划分数，转移只需要枚举以第 i 个字符结尾的所有回文串

$$dp[i] = 1 + \min_{s[j+1..i] \text{ 为回文串}} dp[j]$$

由于一个字符串最多会有 $O(n^2)$ 个回文子串，因此上述算法的时间复杂度为 $O(n^2)$，无法接受，为了优化转移过程，下面给出一些引理。

记字符串 s 长度为 i 的前缀为 $pre(s,i)$，长度为 i 的后缀为 $suf(s,i)$。

周期：若 $0 < p \leq |s|$，$\forall 1 \leq i \leq |s| - p, s[i] = s[i+p]$，就称 p 是 s 的周期。

border：若 $0 \leq r < |s|$，$pre(s,r) = suf(s,r)$，就称 $pre(s,r)$ 是 s 的 border。

周期和 border 的关系：t 是 s 的 border，当且仅当 $|s| - |t|$ 是 s 的周期。

> 证明：
>
> 若 t 是 s 的 border，那么 $pre(s,|t|) = suf(s,|t|)$，因此 $\forall 1 \leq i \leq |t|, s[i] = s[|s| - |t| + i]$，所以 $|s| - |t|$ 就是 s 的周期。
>
> 若 $|s| - |t|$ 为 s 周期，则 $\forall 1 \leq i \leq |s| - (|s| - |t|) = |t|, s[i] = s[|s| - |t| + i]$，因此 $pre(s,|t|) = suf(s,|t|)$，所以 t 是 s 的 border。

引理 1：t 是回文串 s 的后缀，t 是 s 的 border 当且仅当 t 是回文串。

> 证明:
>
> 对于 $1 \leq i \leq |t|$，由 s 和 t 为回文串，因此有 $s[i] = s[|s| - i + 1] = s[|s| - |t| + i]$，所以 t 是 s 的 border。
>
> 对于 $1 \leq i \leq |t|$，由 t 是 s 的 border，有$s[i] = s[|s| - |t| + i]$，由 s 是回文串，有 $s[i] = s[|s| - i + 1]$，因此 $s[|s| - i + 1] = s[|s| - |t| + i]$，所以 t 是回文串。

下图中，相同颜色的位置表示字符对应相同。

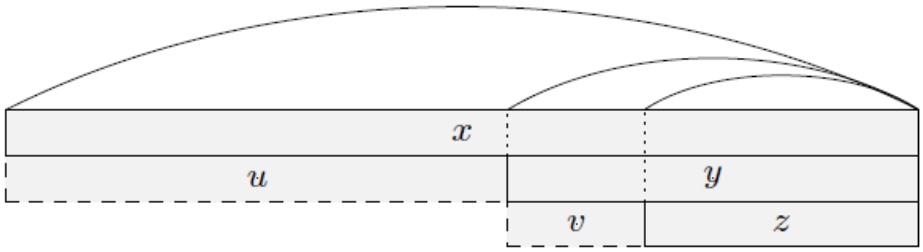
引理 2：t 是回文串 s 的 border ($|s| \leq 2|t|$)，s 是回文串当且仅当 t 是回文串。

> 证明：
>
> 若 s 是回文串，由引理 1，t 也是回文串。
>
> 若 t 是回文串，由 t 是 s 的 border，因此 $\forall 1 \leq i \leq |t|, s[i] = s[|s| - |t| + i] = s[|s| - i + 1]$，因为 $|s| \leq 2|t|$，所以 s 也是回文串。

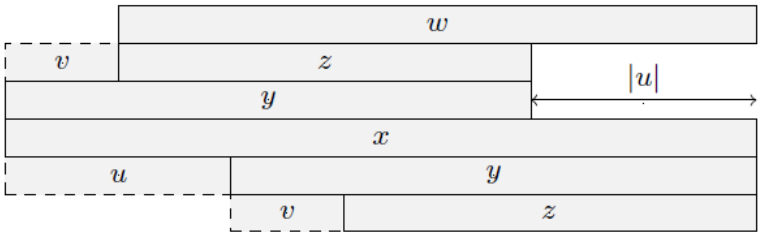
引理 3：t 是字符串 s 的 border，则 $|s| - |t|$ 是 s 的周期，$|s| - |t|$ 为 s 的最小周期，当且仅当 t 是 s 的最长回文真后缀。

引理 4：x 是一个回文串，y 是 x 的最长真回文后缀，z 是 y 的最长回文真后缀。令 u, v 分别为满足 $x = uy, y = vz$ 的字符串，则有下面三条性质

1. $|u| \geq |v|$;
2. 如果 $|u| > |v|$，那么 $|u| > |z|$;
3. 如果 $|u| = |v|$，那么 $u = v$。



> 证明：
>
> 1. 由引理 3 的推论，$|u| = |x| - |y|$ 是 x 的最小周期，$|v| = |y| - |z|$ 是 y 的最小周期。考虑反证法，假设 $|u| < |v|$，因为 y 是 x 的后缀，所以 u 既是 x 的周期，也是 y 的周期，而 $|v|$ 是 y 的最小周期，矛盾。所以 $|u| \geq |v|$。
> 2. 因为 y 是 x 的 border，所以 v 是 x 的前缀，设字符串 w，满足 $x = vw$（如下图所示），其中 z 是 w 的 border。考虑反证法，假设 $|u| \leq |z|$，那么 $|zu| \leq 2|z|$，所以由引理 2，w 是回文串，由引理 1，w 是 x 的 border，又因为 $|u| > |v|$，所以 $|w| > |y|$，矛盾。所以 $|u| > |z|$。
> 3. u, v 都是 x 的前缀，$|u| = |v|$，所以 $u = v$。



推论：s 的所有回文后缀按照长度排序后，可以划分成 $\log |s|$ 段等差数列。

> 证明：
>
> 设 s 的所有回文后缀长度从小到大排序为 $l_1, l_2, \ldots, l_k$。对于任意 $2 \leq i \leq k-1$，若 $l_i - l_{i-1} = l_{i+1} - l_i$，则 l_{i-1}, l_i, l_{i+1} 构成一个等差数列。否则 $l_i - l_{i-1} \neq l_{i+1} - l_i$，由引理 4，有 $l_{i+1} - l_i > l_i - l_{i-1}$，且 $l_{i+1} - l_i > l_{i-1}$，$l_{i+1} > 2l_{i-1}$。因此，若相邻两对回文后缀的长度之差发生变化，那么这个最大长度一定会相对于最小长度翻一倍。显然，长度翻倍最多只会发生 $O(\log |s|)$ 次，也就是 s 的回文后缀长度可以划分成 $\log |s|$ 段等差数列。
>
> 该推论也可以通过使用弱周期引理，对 s 的最长回文后缀的所有 border 按照长度 x 分类，$x \in [2^0, 2^1), [2^1, 2^2), \ldots, [2^k, n)$，考虑这 $\log |s|$ 组内每组的最长 border 进行证明。详细证明可以参考金策的《字符串算法选讲》和陈孙立的 2019 年 IOI 国家候选队论文《子串周期查询问题的相关算法及其应用》。

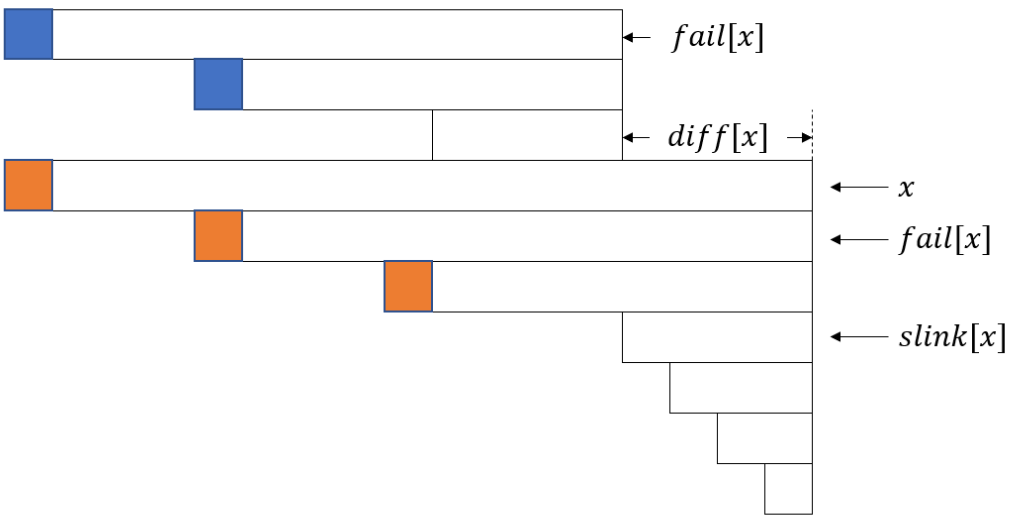
有了这个结论后，我们现在可以考虑如何优化 dp 的转移。

回文树上的每个结点 u 需要多维护两个信息，$diff[u]$ 和 $slink[u]$。$diff[u]$ 表示结点 u 和 $fail[u]$ 所代表的回文串的长度差，即 $len[u] - len[fail[u]]$。$slink[u]$ 表示 u 一直沿着 fail 向上跳到第一个结点 v，使得 $diff[v] \neq diff[u]$，也就是 u 所在等差数列中长度最小的那个结点。

根据上面证明的结论，如果使用 `slink` 指针向上跳的话，每向后填加一个字符，只需要向上跳 $O(\log |s|)$ 次。因此，可以考虑将一个等差数列表示的所有回文串的 dp 值之和（在原问题中指 $\min$），记录到最长的那一个回文串对应结点上。

$g[v]$ 表示 v 所在等差数列的 dp 值之和，且 v 是这个等差数列中长度最长的结点，则 $g[v] = \sum_{slink[x]=v} dp[i - len[x]]$。

下面我们考虑如何更新 g 数组和 dp 数组。以下图为例，假设当前枚举到第 i 个字符，回文树上对应结点为 x。$g[x]$ 为橙色三个位置的 dp 值之和（最短的回文串 $slink[x]$ 算在下一个等差数列中）。$fail[x]$ 上一次出现位置是 $i-diff[x]$（在 $i-diff[x]$ 处结束），$g[fail[x]]$ 包含的 dp 值是蓝色位置。因此，$g[x]$ 实际上等于 $g[fail[x]]$ 和多出来一个位置的 dp 值之和，多出来的位置是 $i-(slink[x]+diff[x])$。最后再用 $g[x]$ 去更新 $dp[i]$，这部分等差数列的贡献就计算完毕了，不断跳 $slink[x]$，重复这个过程即可。具体实现方式可参考例题代码。



最后，上述做法的正确性依赖于：如果 x 和 $fail[x]$ 属于同一个等差数列，那么 $fail[x]$ 上一次出现位置是 $i-diff[x]$。

> 证明：
>
> 根据引理 1，$fail[x]$ 是 x 的 border，因此其在 $i-diff[x]$ 处出现。
>
> 假设 $fail[x]$ 在 $(i-diff[x],i)$ 中的 j 位置出现。由于 x 和 $fail[x]$ 属于同一个等差数列，因此 $2|fail[x]|\geq x$。多余的 $fail[x]$ 和 $i-diff[x]$ 处的 $fail[x]$ 有交集，记交集为 w，设串 u 满足 $uw=fail[x]$。用类似引理 1 的方式可以证明，w 是回文串，而 x 的前缀 $s[i-len[x]+1..j]=uwu$ 也是回文串，这与 $fail[x]$ 是 x 的最长回文前缀（后缀）矛盾。

例题

给定一个字符串 s，要求将 s 划分为 $t_1,t_2,\ldots,t_k$，其中 k 是偶数，且 $t_i=t_{k-i}$，求这样的划分方案数，答案对 10^9+7 取模。

其中 $2\leq|s|\leq 10^6$。

来源：Codeforces 932G Palindrome Partition。

构造字符串 $t=s[0]s[n-1]s[1]s[n-2]s[2]s[n-3]\ldots s[n/2-1]s[n/2]$，问题等价于求 t 的偶回文划分方案数，把上面的转移方程改成求和形式并且只在偶数位置更新 dp 数组即可。

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int mod = 1e9 + 7;
const int maxn = 1000000 + 5;

inline int add(int x, int y) {
    x += y;
    return x >= mod ? x -= mod : x;
```

```
}

namespace pam {
    int sz, tot, last;
    int ch[maxn][26], len[maxn], fail[maxn];
    int cnt[maxn], dep[maxn], dif[maxn], slink[maxn];
    char s[maxn];
    int node(int l) {
        sz++;
        memset(ch[sz], 0, sizeof(ch[sz]));
        len[sz] = l;
        fail[sz] = 0;
        cnt[sz] = 0;
        dep[sz] = 0;
        return sz;
    }
    void clear() {
        sz = -1; last = 0;
        s[tot = 0] = '$';
        node(0); node(-1);
        fail[0] = 1;
    }
    int getfail(int x) {
        while (s[tot - len[x] - 1] != s[tot]) x = fail[x];
        return x;
    }
    void insert(char c) {
        s[++tot] = c;
        int now = getfail(last);
        if (!ch[now][c - 'a']) {
            int x = node(len[now] + 2);
            fail[x] = ch[getfail(fail[now])][c - 'a'];
            dep[x] = dep[fail[x]] + 1;
            ch[now][c - 'a'] = x;

            dif[x] = len[x] - len[fail[x]];
            if (dif[x] == dif[fail[x]]) slink[x] = slink[fail[x]];
            else slink[x] = fail[x];
        }
        last = ch[now][c - 'a'];
        cnt[last]++;
    }
}
using pam::len;
using pam::fail;
using pam::slink;
using pam::dif;

int n, dp[maxn], g[maxn]; char s[maxn], t[maxn];

int main() {
    pam::clear();
    scanf("%s", s + 1);
    n = strlen(s + 1);
    for (int i = 1, j = 0; i <= n; i++) t[++j] = s[i], t[++j] = s[n - i +
1];
    dp[0] = 1;
    for (int i = 1; i <= n; i++) {
```

```
        pam::insert(t[i]);
        for (int x = pam::last; x > 1; x = slink[x]) {
            g[x] = dp[i - len[slink[x]] - dif[x]];
            if (dif[x] == dif[fail[x]]) g[x] = add(g[x], g[fail[x]]);
            if (i % 2 == 0) dp[i] = add(dp[i], g[x]);
        }
    }
    printf("%d", dp[n]);
    return 0;
}
```

参考资料

- EERTREE: An Efficient Data Structure for Processing Palindromes in Strings;
- Palindromic tree;
- 2017 年 IOI 国家候选队论文集 回文树及其应用 翁文涛;
- 2019 年 IOI 国家候选队论文集 子串周期查询问题的相关算法及其应用 陈孙立;
- 字符串算法选讲 金策;
- A bit more about palindromes;
- A Subquadratic Algorithm for Minimum Palindromic Factorization。

例题

例题 1

给定一个空串，有 n 次操作，每次会在字符串左端或右端加一个字符，求每次操作后的字符串有多少个 `border`。

其中 $1 \leq n \leq 10^6$。

来源：2019牛客暑期多校训练营（第十场） C. Gifted Composer。

例题 2

求长度小于等于 n，字符集大小为 k 的串中，有多少个存在弱双回文划分，答案对 998244353 取模。弱双回文划分指一个串可以划分左右两个部分，使得两部分都是回文串，且最多只能有一个部分是空串。

其中 $1 \leq n \leq 10^5, 1 \leq k \leq 26$。

来源：ICPC 2019-2020 North-Western Russia Regional Contest D. Double Palindrome。

例题 3

给定一棵带边权 $\{0, 1\}$ 的无根树，求回文路径的个数。

其中 $3 \leq n \leq 5 \times 10^4$。

来源：yww 与树上的回文串。

例题 4

给定两个数字串 a，b，分别从两个串里面挑一个子序列 x 和 y，将两个子序列表示成 1000 进制的整数，求有多少对子序列满足 $x > y$，两对子序列不同当且仅当存在一个位置不同，答案对 998244353 取模。

来源：2019 CCPC Qinhuangdao Onsite C. Sakurada Reset。

最后

"后缀结构" 对算法竞赛中常见的字符串算法和问题进行了全面的展示。

字符串专题中最重要也是最基础的就是字符串匹配的相关的问题，本文从 3 个字符串匹配的问题出发，有浅入深地介绍了相关解决算法。1 个模板串在 1 个文本串里匹配，引出字符串匹配自动机，而 Knuth–Morris–Pratt 算法是字符串匹配自动机压缩空间的精简实现，同时 KMP 算法的后缀链接有一些很好的性质，在例题中得到了体现；多个模板串同时在 1 个文本串匹配，引入了字典树的结构，将 Knuth–Morris–Pratt 算法在字典树上进行扩展，得到了 Aho–Corasick 算法，将完整的建出了所有转移边的自动机称为 Trie 图，使得我们将一些字符串问题转化成图论问题加以解决。最后，1 个文本串，回答多个模板串在文本串中的出现次数，引入了后缀结构，从后缀树的角度，阐述了后缀数组和后缀自动机这两种后缀结构。

在本文中还穿插了一些其他的字符串算法和问题，求每个文本串每个后缀和模板串的最长公共前缀的 Z 算法，暴力压位的 Shift-And 算法，结合字符串哈希的 Rabin-Karp 算法，回文串的 Manacher 算法和回文树算法，并对最小回文划分问题的论文相关内容进行了介绍。最后，例题 1 展示了字符串的周期和重复问题，本文并未具体展开，这部分将会涉及 Period Theorem，Runs，Lyndon Word 的相关结构，难度较大；例题 2，则是一个与数论和组合计数相关的回文串问题，本题涉及了一些字符串的组合性质，难度较大；例题 3，与树论紧密结合，是一个比较麻烦的字符串综合问题；例题 4 简要介绍了一道子序列问题，使用子序列自动机进行动态规划。本文没有涉及到循环串的最小表示法，感兴趣的读者可以自行了解。

例题选择自一些经典问题，2019 年现场赛和训练中出现的题以及笔者觉得比较有趣的问题进行分享，希望字符串的后缀结能够构对大家有所帮助w。

颜俊梁

2017级 计算机科学与工程学院

2020-2-24