# Trivial String Techniques

Falsyta

Peking University

2019 年 1 月 26 日

既然是 Techniques 向的内容那么就从实际问题来谈一些问题的处理方法，大部分方法都是我觉得有一定价值可以用来解决问题的。并且大部分内容还是在 OI 题目中出现过的。
之所以选择不讲新的算法也非常简单，一是很多新算法都是用来解决具体问题的，并且相关的证明都相对复杂，一节课的时间里理解有些困难，另外一方面来说……后缀仙人掌后缀平衡树 border tree 什么的你用过么？个人认为在大部分时间里处理字符串问题的主力还是 SA/SAM ，很少有复杂的字符串算法不依靠 SA/SAM 就做下去的。
很抱歉的一点是，这里提到的一些做法是未经过验证的，所以可能会有疏忽的地方，如果有的话还请谅解。
另外讲课过程中准备了三个向大家提问的问题 (有两个都是已有题解的题目)，上台回答正确的同学之后可以找我 (QQ: 2641127912) 来领一个 Steam 上的 Symphonic Rain 重置版 (112 元)。

# Notations

$s[l:r]$ 表示字符串 $s$ 从第 $l$ 个字符到第 $r$ 个字符的子串（从 1 开始标号），$|s|$ 表示 $s$ 的长度。
当 $l=1$ 时 $s[l:r]$ 简写为 $s[:r]$，表示 $s$ 的一个前缀。当 $r=|s|$ 时 $s[l:r]$ 简写为 $s[l:]$，表示 $s$ 的一个后缀。
$st, s+t$ 表示两个字符串 $st$ 的拼接，$s^k$ 表示 $k$ 个 $s$ 拼起来，特别地，$s^\infty$ 表示 $s$ 的无限循环。
以下内容默认大家对 KMP/SA/SAM 都比较熟悉。

# SAM & LCT

这个方法相对比较局限，但是它能处理的问题一般不太容易用其它方法求解，理解起来也比较简单，所以还是值得一提的。
当我们询问子串的信息的时候，我们可以用扫描线从右向左加入字符，然后用线段树维护当右端点为 $r$ 的时候的答案。
我们考虑令 $y_u$ 表示 $u$ 的子树中最新一次加入的后缀是 $s[y_u:]$，那么我们发现维护 $y_u$ 的过程和 Link-Cut Tree 中 Access 的过程是一致的，于是我们每次新加入一个后缀节点的时候只会引起 $O(\log n)$ 段 $y_u$ 的变动。
那么如果说 $y_u$ 的变化和线段树上维护答案的过程有密切的联系的话我们就可以用这 $O(\log n)$ 段变动来在线段树上维护答案。
我们用一个题目作为例子来说明该具体如何处理。

# Explanation

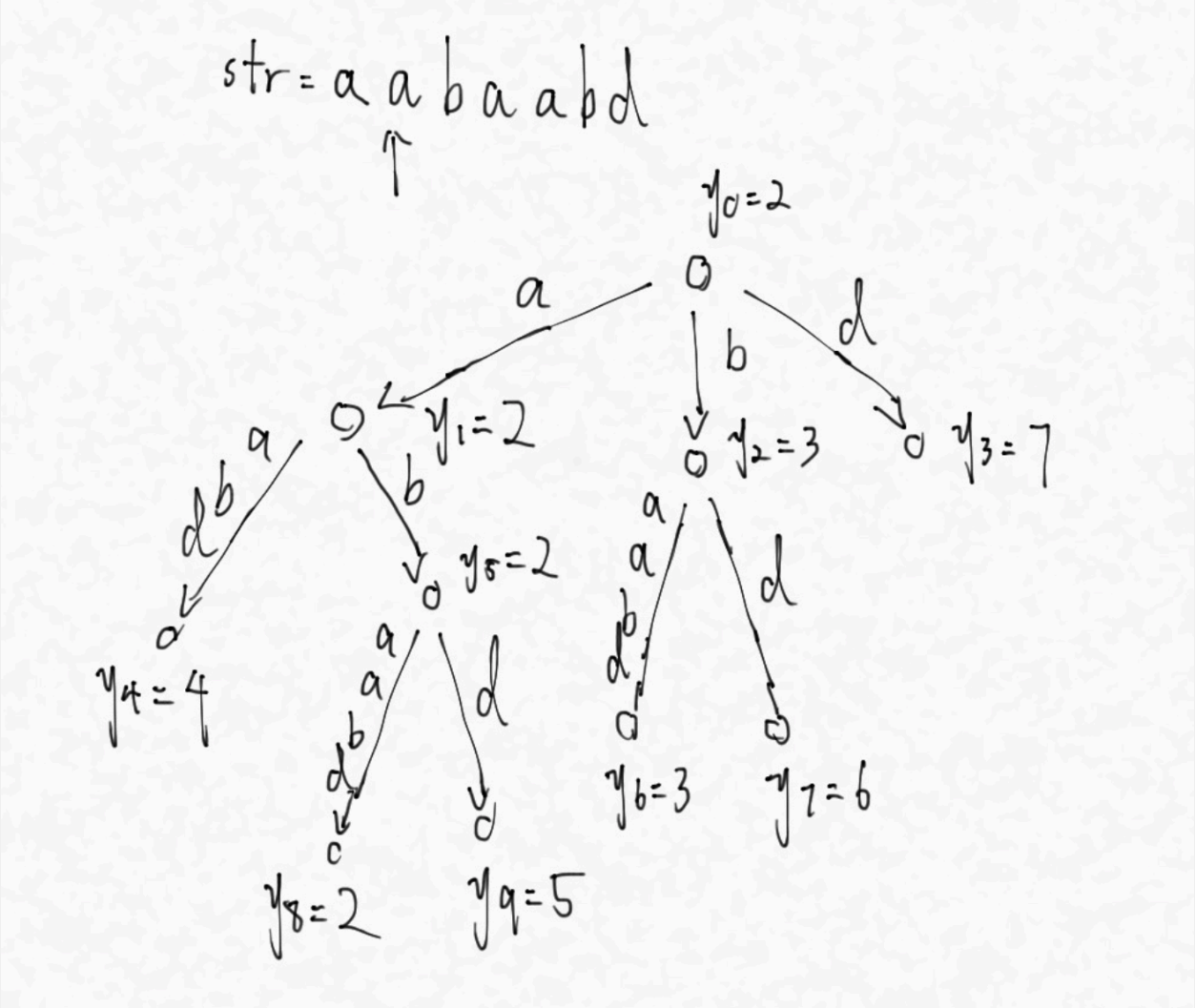


图: 处理到 $i = 2$ 的图示 (节点编号是随便标的)

# Explanation

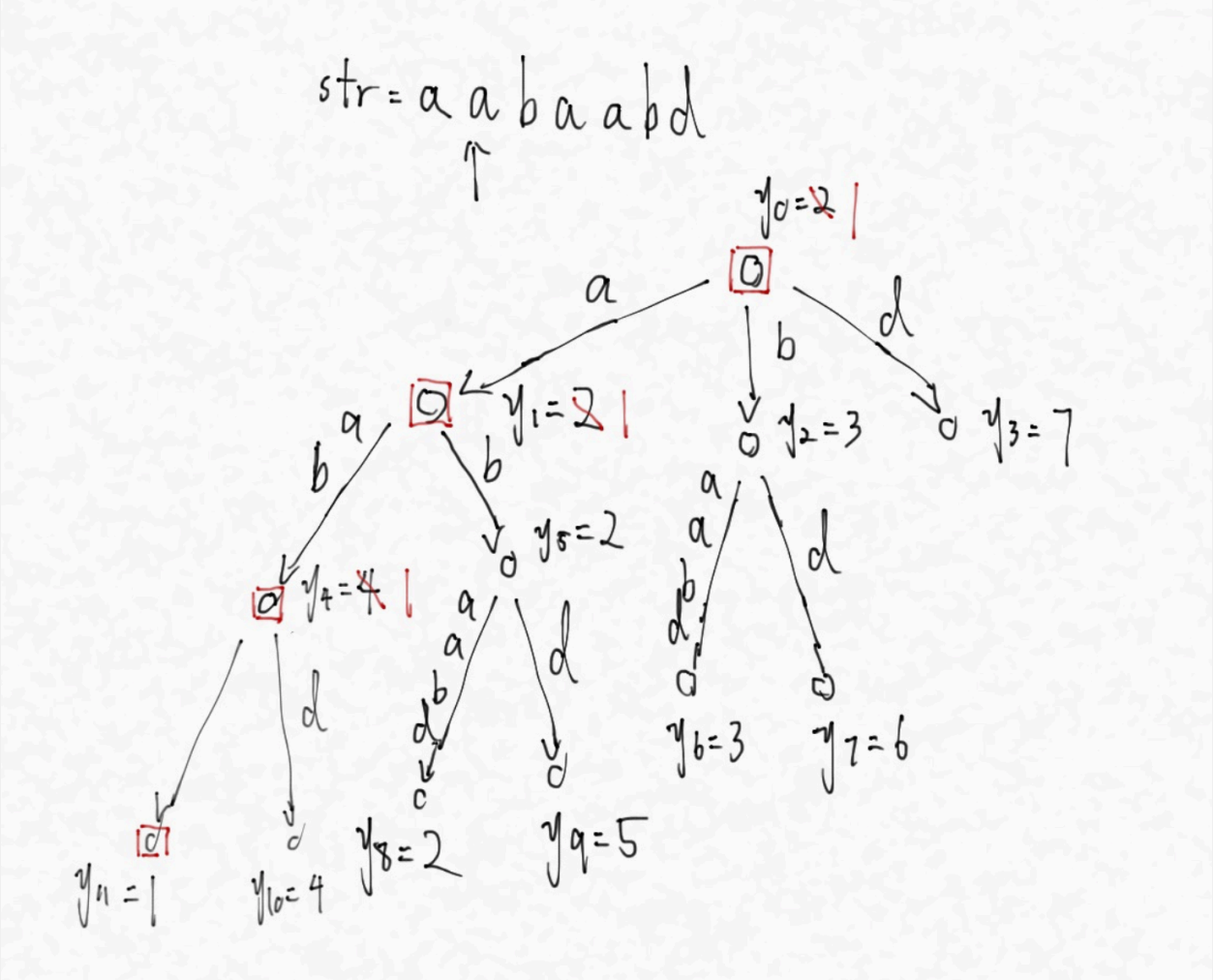


图: 处理到 $i = 1$ 的图示 (节点编号是随便标的)

# Q1

对于给定的字符串 $s$ ， $O(n \log^2 n)$ 预处理 $O(\log n)$ 回答 $s[l:r]$ 的本质不同的子串数。

# A1

在加入 $s[l]$ 时，考虑 $s[l:r]$ 的上一次出现是 $s[k:k+r-l]$ ，那么应该在线段树上 $[r, k+r-l)$ 这个区间上 $+1$ 。
那么假设路径 $(x, y)$ 上的点都满足 $y_u = z$ ，且 $len[par[x]] = a, len[y] = b$ ，那么对线段树的贡献是一个类似平行四边形的加法，修改起来比较显然。
查询 $s[l:r]$ 的本质不同的子串数的话在加入 $s[l]$ 后的线段树查第 $r$ 个位置即可。

# Q2

对于给定的字符串 $s$ ，$O(n\log^2 n)$ 预处理 $O(\log n)$ 回答 $s[l:r]$ 的后缀树节点数。
Source: 《后缀树结点数》命题报告及一类区间问题的优化陈江伦

# A2

我们统计进去的节点要么在 $s[l:r]$ 的后缀树上有两个儿子要么是 $s[k:r](k \geq l)$ 。
先统计前者，注意到加入 $s[l]$ 时，该后缀节点到根的路径上的每一段 $y_u$ 中，这一段中只有最底下的节点 $v$ 会有影响，假设 $v$ 的两个儿子的最早出现位置是 $s[l:l+len[v]-1]$ 和 $s[k:k+len[v]-1]$ ，那么在 $r \geq k+len[v]-1$ 的询问中 $v$ 是一个有两个儿子的节点。
那么按照我们刚才的方法维护就行了。接下来就是如何处理既有两个儿子也是 $s[k:r](k \geq l)$ 的情形。
注意到如果 $s[k:r]$ 有两个儿子那么它的后缀也有两个儿子。于是我们可以二分 $s[k:r]$ ，看 $s[k:r]$ 在后缀树中对应的节点是否满足条件即可。
于是就能去除两个条件的重复了。

# Q3

这个 idea 比较简单就直接放在一个题目里说了。
维护一个字符串 $s$，支持在 $s$ 的某个位置插入一个字符串，删除 $s$ 某一段的字符串。
可以看成是一个普通的文本编辑器。
由于在文本编辑器里进行搜索的时候我们都是一个一个字母键入的，为了模拟这一过程，每次询问给出一个询问串 $p$，对于每一个 $1 \leq i \leq |p|$，你需要回答 $p[:i]$ 在 $s[l:r]$ 中出现了多少次。
只需要三个操作都比暴力快即可 ($o(|s|)$)。

# A3

在字符串中间插入比较困难因此我们选择块状链表。问题是如何快速回答询问。
维护块内的询问是比较简单的但是如何维护块之间的询问是问题。显然每个 $|p| \geq L$ 的询问我们可以暴力处理。因此我们只考虑比较短的询问串。
我们维护出每个块断点前后 $L-1$ 个字符组成的字符串 $r_i = s[pos_i - L + 1 : pos_i + L - 2]$ 。可以看出对于比较短的询问串 $p$ 我们只需要考虑其在 $r_i[L - |p| + 1 : L + |p| - 1]$ 中出现了多少次就行了。于是把所有 $r_i$ 建 Trie 上 SAM 。
这可以看成是一个（$k$ 是出现位置的左端点）$l \leq k \leq r - |p| + 1, |k - pos_i| < |p|, L_p \leq dfn(s[k:]) \leq R_p$ 的三维数点（第三维是 dfs 序，保证 $p$ 在 $s[k:]$ 中出现了）

# Border

定义 $\text{border}(s) = \{s[:i] | s[:i] = s[|s|-i+1:]\}$ ，$\text{Lborder}(s)$ 是 $s$ 最长的 border 。
若有 $1 \le p \le |s|$ ，满足 $\forall 1 \le i \le |s| - p, s[i] = s[i+p]$ 称 $p$ 是 $s$ 的一个周期 (period) 。
Periodicity Lemma: 若 $p, q$ 都是串 $s$ 的周期且 $p + q \le |s| + \gcd(p, q)$ ，则 $\gcd(p, q)$ 也是 $s$ 的一个周期。
如果 $\text{Lborder}(s) \ge |s|/2$ ，那么 $s$ 有一个 $|s| - \text{Lborder}(s)$ 的周期。
因此 $s$ 的所有 border 的下标会形成 $O(\log n)$ 个等差数列。

# KMP

KMP 算法可以求出 $s[1:i]$ 的 lborder 。

```
def initkmp(s):
    n = len(s)
    j = -1
    Next = [j]
    for i in range(s):
        while j >= 0 and s[j + 1] != s[i]: j = Next[j]
        if s[j + 1] == s[i]: j += 1
        Next.append(j)
    return Next
```

# Q4

给出一个字符串 $s$ ， $O(n \log n)$ 预处理， $O(1)$ 回答 $s[l:r]$ 的最大后缀。
Source: 字符串算法选讲金策

# A4

先考虑最小后缀怎么求，我们先求 $t = \min\{s[i:] | l \le i \le r\}$ ，可以发现答案一定是 $t$ 的最短 border 。于是
$\text{minsuf}(s[l:r]) = \min(t, \text{minsuf}(s[r-m+1:r]))(m = \lfloor \lg(r-l+1) \rfloor)$ 。对每个 $r, m$ 都预处理即可。
类似地，我们令
$\text{maxsuf}(s[l:r]) = \max(t, \text{maxsuf}(s[r-m+1:r]))(m = \lfloor \lg(r-l+1) \rfloor)$ ，接下来的问题就是怎么求出 $t$ 。
令 $s[i_1:] = \max\{s[i:] | l \le i \le r\}, s[i_2:] = \max\{s[i:] | l \le i < i_1\}$ ，容易看出如果 $s[i_1:] \ne t$ ，那么由于 $s[i_1:]$ 是 $s[i_2:]$ 的前缀，$s[i_2:]$ 是 $t$ 的前缀因此 $t$ 有 $i_2 - i_1$ 的周期。于是只要求最长公共后缀得出周期延伸至何处即可。

# Q5

定义 $\mathrm{suf}(s) = \{s[i:] | 1 \le i \le |s|\}$ 同理 $\mathrm{pre}(s) = \{s[:i] | 1 \le i \le |s|\}$ 。
定义 $\tau(s,t) = \mathrm{suf}(s) \cap \mathrm{pre}(t)$ ，给出 $n$ 个字符串 $s_1, s_2, \ldots s_n$ ，$O(m \log n)$ 预处理 $O(\log n)$ 回答 $s_i[k:]$ 是否在 $\tau(s_i, s_j)$ 中。

# A5

我们只需要找到 $\tau(s,t)$ 中最长的字符串 $p_1$ ，其余的字符串都是 $p_1$ 的 border 。根据之前的定理 $p_1$ 的 border 形成了 $O(\log n)$ 个等差数列容易检查。
找 $p_1$ 可以把所有 $s_i$ 的 Trie 建成 SAM 把所有 $s_i$ 的后缀都放到 set 里让 $s_j$ 代表的后缀节点查询前驱后继即可。

# Q6

$O(n)$ 预处理 $O(1)$ 时间内回答 $s$ 是否在 $s[:i]s[j:]$ 中出现了。

# A6

即回答是否存在 $x \in \text{border}(\text{s}[:\text{i}]), y \in \text{border}(\text{s}[\text{j}:])$ 使得 $x + y = |s|$ 。
注意到 $x \geq |s[:i]|/2$ 和 $y \geq |s[j:]|/2$ 必有其一成立，不妨设前者成立，令 $d$ 是 $s[:i]$ 的 period ，故 $x = i - kd$ 。
我们的目标即寻找一个 $k$ 使得 $s[i - kd : m]$ 是 $s[j : m]$ 的前缀。于是我们求 $i_1 = \text{LCP}(s, s[:d]^\infty), i_2 = \text{LCP}(s[1:i]s[j:m], s[:d]^\infty)$ 。
容易看出 $i_1 + kd = i_2$ 或 $i_1 = |s| \leq i_2 - kd$ ，求 $i_1, i_2$ 是 $O(1)$ 的，故 $k$ 也容易求出。

# Compressed Pattern Matching

母串 $s$ 是由一个若干个字典串 $d_{a_1}, d_{a_2}, \ldots d_{a_n}$ 拼接而成的，模板串 $p$ 是直接给出的，求 $p$ 在 $s$ 中是否出现/出现次数。
Simplified: 字典串是以字符串序列的形式直接给出的。
LZW compress: 字典串是以 Trie 的形式给出的。
这里我们只讨论 Simplified 的情形，LZW compressed 留给读者自行思考。

# Our Method

我们考虑直接把 KMP 算法拉过来改一改，我们的流程是：找到失配位置-> 跳到正确的 next 位置继续匹配-> 找到失配位置->...
找到失配位置只需要用 SA/SAM 查 LCP 即可。主要问题是如何找到正确的 next 位置。

```
def getfail(i, j):
    if s[i+1] == p[j]:
        if j == len(p): return calcans(i+1)
        else: return i+1, j+1, 0
    if Next[j] < j/2: return getfail(i, Next[j]) # (1)
    l = j - Next[j]
    j = Next[j]
    if s[i+1] != p[j]: return getfail(i, l + (j % l)) # (2)
    else:
        k = j + LCP(s[i+1:], p[j:])
        if k > len(p): return calcans(i + k - j)
        elif k - Next[k] != l:
            return i + k - j - 1, k - 1, 0 # (3)
        else:
            return getfail(i+k-j-1, (k-1)%l+l) # (2)
```

# Analysis

我们注意到 (1),(2) 都会把 $j$ 变为最多原来的 $2/3$ ，因此这两部分最多是 $O(\log mn)$ 的。
下面我们考虑 (3) 部分的复杂度，我们令 $b_i = [\text{next}_i \geq i/2]$ ，将 $b$ 序列中从左到右第 $i$ 段连续的 1 称为第 $i$ 层（容易看出来同一层的 period 是相同的）。令第 $i$ 层的周期为 $per_i$ ，有 $\frac{3}{2}|per_i| \leq |per_{i+1}|$ 。
证明：令 $pos_i$ 表示第 $i$ 层最右边的位置，那么显然 $per_i$ 和 $per_{i+1}$ 都是 $S[:pos_i]$ 的周期。如果 $|per_i| + |per_{i+1}| \leq pos_i$ ，那么 $\gcd(|per_i|, |per_{i+1}|)$ 也是 $S[:pos_i]$ 的周期。注意到 $\gcd(|per_i|, |per_{i+1}|)$ 整除 $|per_{i+1}|$ ，这与 $per_{i+1}$ 是第 $i+1$ 层的最短周期矛盾，故我们得到 $|per_i| + |per_{i+1}| > pos_i$ 。又有 $pos_i \geq 2|per_i|$ ，联立两式可得 $\frac{3}{2}|per_i| \leq |per_{i+1}|$ 。
我们可以看出 (2) 是在后退一层而 (3) 是在前进一层，都不会在原有的层里停留，而层数只有 $O(\log n)$ 层，后退的次数是 $O(\log n)$ 次因此前进的次数也是 $O(\log n)$ 次。

# Gawrychowski(2013)

由于线性的做法比较复杂，我们在这里只介绍 $O(n \log n)$ 的部分，要注意的是，这个算法只负责判断 $p$ 是否在 $s$ 中出现了。

```
l = p[:l] is the longest prefix of p that is suffix of s[1]
k = 2
while k <= n:
    r = p[r:] is the longest suffix of p that is a prefix of s[k]
    check for an occurance of p in p[:l]p[r:] # (1)
    if p[:l]s[k] is a prefix of p:
        l += len(s[k])
        k += 1
        continue

    b = longest long border of p[:l] that p[:b]s[k] is a prefix of p # (2)
    if b not found:
        l = longest prefix of p that is suffix of p[(l+1)//2:l] # (3)
        continue

    l = b + s[k]
    k += 1
```

# Details

(1): 我们所关心的出现起始位置在 $p[:l]$ 中，出现在 $s[k:]$ 的情形会被接下来处理
(2): 这里的处理方式 (1) 是类似的，令 $p[:l]$ 的周期为 $d$ 则 $b = kd$ ，我们处理出 $i_1 = \mathrm{LCP}(p[:l]s[k], p[:d]^\infty), i_2 = \mathrm{LCP}(\mathrm{p}, \mathrm{p}[:\mathrm{d}]^\infty)$ 则 $i_1 + kd \leq i_2$ 。
(3): 应该可以用一次 KMP 处理出来 (参见 NOI2014 动物园)

# Compressed Pattern Matching(Multi Patterns)

母串 $s$ 是由一个若干个字典串 $d_{a_1}, d_{a_2}, \ldots d_{a_n}$ 拼接而成的，有 $m$ 个模板串 $p_i$ ，是直接给出的，求每个 $p_i$ 是否在 $s$ 中出现。
Simplified: 字典串是以字符串序列的形式直接给出的。
LZW compress: 字典串是以 Trie 的形式给出的。
这里我们只讨论 Simplified 的情形，LZW compressed 留给读者自行思考。

# Gawrychowski(2014)

```
P = []
cur = s[1]

for k in range(2,m+1):
    P.append((cur, s[k]))
    cur = getprefix(cur, s[k])

for a, b in P:
    checkoccur(a, b)
```

和刚才的做法差不多，getprefix 用来在 $p_i$ 的 AC 自动机里定位 $cur + s_k$ ，checkoccur 检查 $cur + s_k$ 中各个 $p_i$ 的出现情况。

# Details

getprefix: 这个定位工作就是查询 $cur + s_k$ 的一个最大后缀使得其是某个 $p_i$ 的前缀。都翻转过来就是查询 $s_k^R + cur^R$ 的一个最大前缀使得其是某个 $p_i^R$ 的后缀，那么就是查询 $s_k^R + cur^R$ 在 $p_i^R$ 的 Trie 上 SAM/SA 的最长匹配。类似的问题在 SPOJ COT4 上出现过，可以用二分 hash/在 SA 上二分进行定位。
checkoccur: 我们把所有 $cur, s_k$ 离线下来做，我们对于每个 $p_i$ 枚举在 $a$ 中出现的部分 $p_i[:j]$ ，那么就是要求 $p_i[:j]$ 是 $cur$ 的后缀，$p_i[j+1:]$ 是 $s_k$ 的前缀，也就是说 $cur^R$ 在 $p_i^R$ 的 Trie 上 SAM 的 $p_i[:j]^R$ 对应节点的子树里且 $s_k$ 在 $p_i$ 的 Trie 上 SAM 的 $p_i[j+1:]$ 对应节点的子树里，这可以看成是一个二维数点问题，
于是就可以在 $O(n \log n)$ 的时间里解决了。

# Bidirectional Suffix Tree(Inenaga, 2002)

后缀树是可以双向加字符的。这里描述的算法并不是 Inenaga 论文中的原算法(尽管在浏览之后我觉得和论文中描述的应该大致相同)，是 @negviizhao 实现的做法。
向前加字符我们采取类似后缀自动机的方法，向后加字符我们采取 Ukkonen's Algorithm 。考虑到 Ukkonen's Algorithm 的细节并不广为人知，所以还是简要描述一下。
Ukkonen's Algorithm 维护的是一棵隐式后缀树，任何节点都至少有两个儿子。叶节点永远是叶节点，不会随着向后加字符变成非叶节点。而如果 $s[i:]$ 是叶节点，意味着比 $s[i:]$ 长的后缀都是叶节点，也就是说非叶节点是某个后缀 $s[k:]$ 的所有后缀。
在向后加字符的时候，我们从 $s[k:]$ 从长到短枚举尚未加入的后缀，依次看是否需要把这个后缀加到后缀树里。从长到短枚举是用 Suffix Link 实现的，一个节点的 Suffix Link 指向它去掉第一个字符后在树中对应节点的位置。

# Main Idea

Right Extension: 主要问题在向后加字符的时候如何维护后缀自动机的转移边。涉及到的节点主要是加入的新节点和后缀节点之间的转移，以及如果加入了一个非叶结点，需要将一些转移修改为到它，这在找下一个后缀的位置时顺便修改即可。
Left Extension: 向前加字符的时候可能会把一个叶节点变成非叶节点，需要进行处理，结构变动的时候记得更新 Ukkonen's Algorithm 部分维护的当前节点，
具体实现有不少细节问题，请参考
https://paste.ubuntu.com/p/JPDwhZ37fQ/ 。

# QA 7

维护一个字符串 $s$ ，支持 push-back/push-front。
每次操作后输出 $s$ 本质不同的子串数。
Source: CTT2018 Day3 Close
直接套用上面的算法即可。当时我的做法是维护所有叶节点的后缀平衡树，因为叶节点的相对大小关系是不会改变的。后缀平衡树是用二分 hash 的 set 来实现的因此复杂度是 $O(n \log^2 n)$ （唯一的好处是容易实现一点）。

# Warning

请记住，标准的 Ukkonen's Algorithm 会在最后插入终止符 \$ ，而这在双向的算法里是无法做到的。也就是说有一些非叶的后缀节点的信息被丢失了（当它只有一个儿子的时候）。
因此与其说是算法，更多地还只是一个方法而已。
那么怎么去进一步的维护这些非叶的后缀节点的信息呢？答案就是把 $s[k:]$ 递归下去进行维护。
要注意直接递归维护 $s[k:]$ 的复杂度并不正确，但我们注意到如果说 $|s[k:]| \geq |s|/2$ ，说明 $s$ 有一个周期串后缀，我们可以利用周期串的性质进行维护。
我们来通过一个例题说明一下。

# Q8

维护一个字符串 $s$ ，给出询问串的序列 $t_i$，支持 push-back/push-front/查询 $t_i$ 在 $s$ 中的出现次数。
$O(\log n)/O(\log^2 n)$

# A8

由于只有 push-front/push-back 我们用两个后缀平衡树来维护向前加和向后加的字符串，令其为 $s_0$ 和 $s_1$ ，还要一并把所有 $t_i$ 也维护进平衡树。现在要处理如何计算跨越中点的出现次数，不跨越的在平衡树里查就好了。
查最长的 $t_i[:l_0] = s_0[|s_0| - l_1 + 1:]$ 和 $s_1[:|t| - l_1 + 1] = t_i[l_1:]$ ，跨越中点的出现次数即 $t_i$ 在 $t_i[:l_0]t_i[l_1:]$ 中的出现次数。
参照前面的实现即可。
应 用 失 败
上一页说的算法实在是太繁杂了，除非我找到什么很有趣的应用否则应该是不会出出来了。

# Lyndon Words

以下内容是我觉得时间充裕的情况下再去介绍的内容。尽管这一部分的确有一些很强的性质也有实际的应用，但是其推导对于之前没有了解的选手而言比较困难，主要做于科普用途。

Lyndon 串：如果一个串 $s$ 满足 $s = \min\{s[i:]|1 \le i \le |s|\}$ 那么我们称串 $s$ 为 Lyndon 串。

Lyndon 划分：我们可以发现每个串都可以被划分成 $p_1^{q_1} \dots p_k^{q_k} (p_1 > p_2 > \dots > p_k)$ 且 $p_1, p_2, \dots p_k$ 均为 Lyndon 串。

# Answer to Substring Minimal/Maximal Suffix

Lyndon 划分是对这个问题的重要指引。
考虑合并 $u, v$ 的信息得到 $uv$ 的最小后缀。
定义 $\text{minsuf}(u, v) = \min\{u[i:]v | 1 \le i \le |u|\}$ ，显然
$\text{minsuf}(uv) = \min(\text{minsuf}(u, v), \text{minsuf}(v))$ 。
假设 $\text{minsuf}(u, v) = u[i:]v$ ，可以看出只有 $u$ 中的一部分后缀才可能成为 $u[i:]$ ，令其为 $u[i_1:], u[i_2:], \ldots u[i_l:](i_1 < i_2 < \ldots i_l)$ 。容易发现 $u[i_{j+1}:]$ 是 $u[i_j:]$ 的前缀。
令 $s$ 的 Lyndon 划分为 $s = p_1^{q_1} \ldots p_k^{q_k}$ ，可以验证每个 $u[i_j:]$ 都可以被写成 $p_{i'}^{q_{i'}} \ldots p_k^{q_k}$ 。

# Answer to Substring Minimal/Maximal Suffix

引理：若 $u[i_j:] = p_{i'}^{q_{i'}} \dots p_k^{q_k}$ ，则 $u[i_{j+1}]$ 是 $p_{i'}$ 的前缀。
证明思路：反证法，验证不是的话会违反 Lyndon 划分的规则。
由此可以看出 $l = O(\log n)$ 。
求 $i_1, \dots i_l$ 的方法：递归到 $s[r-m+1:r](m = \lfloor \lg(r-l+1) \rfloor)$ 可以求出 $i_2 \dots i_l$ ，而 $i_1$ 可以用之前提到的求最大后缀中求 $t$ 的方法求出（把字母表的大小关系反转情况下的最大后缀）。

# Q9

维护一个字符串，支持把某个区间的字符加上一个数和询问 $s[l:r]$ 的最小后缀是谁。
要求 $O(\text{polylog}(n))$

# A9

在线段树的每个节点上维护出该区间的 $i_1, i_2 \dots i_l$ ，在询问时会有 $O(\log^2 n)$ 个候选后缀进行暴力判断找出最小的。

# Thanks for listening

希望通过今天的一些介绍能让大家对字符串算法在当今的发展有更多的认识，即使是在今天，仍然不断的有重要的新成果被发表。字符串领域仍然有很多未知的问题在等待着我们。
希望各位享受在冬令营的学习时光，在考试中能有理想的发挥。
感谢 CCF 给我来冬令营与大家分享交流的机会。
感谢各位老师同学的支持。

# 参考文献

《后缀树结点数》命题报告及一类区间问题的优化陈江伦
《Fim 4》命题报告吴瑾昭
字符串算法选讲金策
维护双向加字符的字符串的隐式后缀树 negiizhao
Optimal pattern matching in LZW compressed strings, Pawe Gawrychowski
Simple and efficient LZW-compressed multiple pattern matching, Pawe Gawrychowski
Bidirectional Construction of Suffix Trees, Shunsuke Inenaga
Minimal Suffix and Rotation of a Substring in Optimal Time, Tomasz Kociumaka