字符串

4182_543_731

2025/07

目录

目录

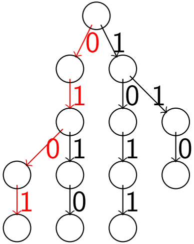
记号约定 & 前言

- S, T 等大写符号通常表示字符串。n 表示字符串长度，默认为 10^5 级别。
- Σ 表示字符集，$|\Sigma|$ 表示字符集大小，默认 $|\Sigma| = 26$。
- $S[l : r]$ 表示 S 中下标 l 到 r 的子串。特别地，$S[l :]$ 表示从位置 l 开始的后缀。字符串下标使用 1-base。
- 简写：LCP 表示两个串的最长公共前缀。
- 一些本课件不涉及的内容：如果你发现 Hash 就够用了并且你认为用 Hash 更简洁，那你可以直接 Hash。
- 关于题目来源：所有这个颜色的文字都是可以点击的。
- Hint: Think in **both** directions!

回顾——Trie

定义：给定若干个串，

- 节点构成有向树的结构。
- 每个节点对应一个字符串的前缀，根表示空串。
- 每条边标记一个字符 c，表示儿子对应字符串为父亲对应字符串向后加入 c 的结果。

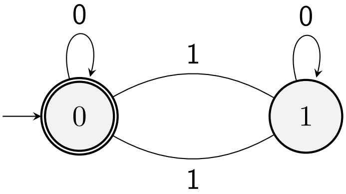


性质：两串的 LCP 正是 Trie 上的 LCA。

什么是一个自动机?

想象你需要判断一个串是否满足某些性质（例如，包含 S/是 S 的子串/…）。

一位一位地把这个串读进来。在这个过程中，我们维护一个**状态**，它记录了读进来的前缀与性质之间的信息。每次读入下一个字符后，我们根据当前状态和这个字符**转移**到下一个状态。这样的状态与转移就被称为自动机。

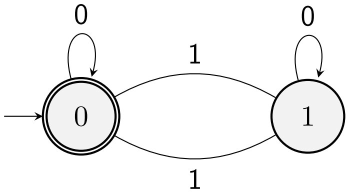


引言——自动机

什么是一个自动机?

想象你需要判断一个串是否满足某些性质（例如，包含 S/是 S 的子串/...）。

一位一位地把这个串读进来。在这个过程中，我们维护一个**状态**，它记录了读进来的前缀与性质之间的信息。每次读入下一个字符后，我们根据当前状态和这个字符**转移**到下一个状态。这样的状态与转移就被称为自动机。



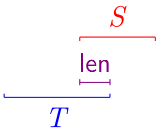
自然的，自动机可以拿来判断一个串是否满足性质。但也有一些时候，构造自动机的过程也揭示了这个性质（也就是 S）的更多特点，使得我们能有更好的关于 S 的结论。

目录

KMP

想象我们读进来一个串 T，需要判定它是否包含了 S 作为子串。那读了 T 的一个前缀后，需要记录什么信息?

如果之后有一个 S 的匹配，那么当前 T 的一个后缀一定能匹配 S 的前缀。



因此，考虑记录最长的长度 len，使得当前 T 这么长的后缀等于 S 这么长的前缀。这自动包含了更短后缀的匹配信息。

KMP——状态

状态为 len，表示当前 T 这么长的后缀匹配 S 这么长的前缀，且这是最长的一组匹配。

然后是转移。在 T 加入下一个字符后，如果上一步这个 len 长度的后缀加上下一个字符还能匹配，那现在 $len+1$，就可以了。[1]

[1]不会存在更长的匹配。因为如果存在，那么去掉最后一个字符就是上一步更长的匹配。

KMP——状态

KMP——状态

状态为 len，表示当前 T 这么长的后缀匹配 S 这么长的前缀，且这是最长的一组匹配。

然后是转移。在 T 加入下一个字符后，如果上一步这个 len 长度的后缀加上下一个字符还能匹配，那现在 $len+1$，就可以了。[1]

否则，我们需要找一个更短的后缀，使得它可以匹配 S 的前缀，同时可以向后扩展这一个字符。因为这个后缀匹配 S 的前缀 $S[1:len]$，问题相当于：

$S[1:len]$ 的后缀中，有哪些可以和 S 的前缀匹配？

我们需要在 S 上找到一些东西来回答这个问题。

[1]不会存在更长的匹配。因为如果存在，那么去掉最后一个字符就是上一步更长的匹配。

KMP——状态

状态为 len，表示当前 T 这么长的后缀匹配 S 这么长的前缀，且这是最长的一组匹配。

$S[1:len]$ 的后缀中，有哪些可以和 S 的前缀匹配?

为此，我们定义 $fail_{len}$ 表示 $S[1:len]$ 的非平凡后缀中，与 S 前缀匹配的最长后缀长度。

那更短的后缀呢？ 长度为 $fail_{len}$ 的后缀匹配了前缀，而更短的后缀也是它的后缀，从而可以匹配到对应前缀上考虑。

进一步可以发现，所有匹配的后缀长度就是沿着 fail 一直跳，跳到的所有长度。

KMP——fail

KMP——状态

状态为 len，表示当前 T 这么长的后缀匹配 S 这么长的前缀，且这是最长的一组匹配。

我们定义 $fail_{len}$ 表示 $S[1:len]$ 的非平凡后缀中，与 S 前缀匹配的最长后缀长度，此时 $S[1:len]$ 的后缀中，与 S 前缀匹配的后缀长度为不断跳 fail 经过的所有长度。

如果有了 fail，那转移是自然的：不断跳 fail，直到当前长度的前缀可以扩展下一个字符，然后跳过去（$+1$）。那么，如何求 fail?

KMP——fail

KMP——状态

状态为 len，表示当前 T 这么长的后缀匹配 S 这么长的前缀，且这是最长的一组匹配。

我们定义 $fail_{len}$ 表示 $S[1:len]$ 的非平凡后缀中，与 S 前缀匹配的最长后缀长度，此时 $S[1:len]$ 的后缀中，与 S 前缀匹配的后缀长度为不断跳 fail 经过的所有长度。

如果有了 fail，那转移是自然的：不断跳 fail，直到当前长度的前缀可以扩展下一个字符，然后跳过去（$+1$）。那么，如何求 fail?

可以看到，fail 的定义和最开始的匹配问题是完全一样的，因此，做法正和匹配相同——从 $fail_{i-1}$ 开始往上跳，直到找到一个能扩展 S_i 的前缀。

每次跳 fail 都减少长度，只有扩展时增加长度。势能分析可知复杂度为 $O(n)$。

KMP

KMP——状态

状态为 len，表示当前 T 这么长的后缀匹配 S 这么长的前缀，且这是最长的一组匹配。

KMP——转移（fail）

定义 $fail_{len}$ 表示 $S[1:len]$ 的非平凡后缀中，与 S 前缀匹配的最长后缀长度。

转移时，不断跳 fail 直到当前前缀可以扩展下一个字符，然后 $+1$。

这样我们就得到了一个判定 T 是否包含 S 的自动机。自然也可以算 S 在 T 中出现次数 (有几次停在整个 S)。

[2]这方面有一些智慧题，但由于篇幅原因+我也没学过

KMP——状态

状态为 len，表示当前 T 这么长的后缀匹配 S 这么长的前缀，且这是最长的一组匹配。

KMP——转移（fail）

定义 $fail_{len}$ 表示 $S[1:len]$ 的非平凡后缀中，与 S 前缀匹配的最长后缀长度。
转移时，不断跳 fail 直到当前前缀可以扩展下一个字符，然后 $+1$。

这样我们就得到了一个判定 T 是否包含 S 的自动机。自然也可以算 S 在 T 中出现次数 (有几次停在整个 S)。

而除了匹配之外，我们构造用到的 fail，也描述了很多 S 自身的性质。对于每个前缀，沿着它的 fail 向上，我们可以找到这个前缀中与前缀匹配的后缀——也被称为 border。[2]

[2]这方面有一些智慧题，但由于篇幅原因+我也没学过

KMP

给定一个串 S，我们构造：

- 状态为 S 的所有前缀。
- 对于每个前缀，fail 指向其最长的匹配某个前缀的后缀。
- 我们隐式地得到了转移边：对于每个前缀，$next_c$ 表示向这个前缀加入字符 c 后，其最长的匹配某个前缀的后缀。

基本应用：单字符串匹配。

现在，考虑把这个东西放到多个 S 上。

AC 自动机

AC 自动机

给定多个串 $S_1, S_2, \ldots$，我们构造：

- 状态为这些串的所有前缀。
- 对于每个节点，fail 指向这个串最长的匹配某个前缀（可以来自其它串）的后缀。
- 也可以得到转移边：对于每个串，$next_c$ 表示向这个串加入字符 c 后，其最长的匹配某个前缀的后缀。

此时状态是所有前缀，因此可以用一个 Trie 表示。KMP 可以看作 Trie 是一条链的特殊情况。那因为 KMP 是一条链的情况，可以考虑自然地扩展到一棵 Trie 树的情况：

AC 自动机——构造

KMP——构造

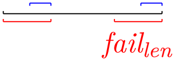
从小到大考虑每个前缀。对于一个前缀 $S[1:i]$，从上一个前缀 $S[1:i-1]$ 开始向上跳 fail (不包含自己)，直到某个前缀能够扩展出 S_i。

自然的扩展：

AC 自动机——构造

按长度从小到大考虑每个前缀（Trie 树上 BFS)。对于一个前缀，从 Trie 树的父亲节点开始向上跳 fail（不包含自己)，直到某个前缀能够扩展出 S_i（在 Trie 树上有这个儿子)。

此时仍然有一个串沿着 fail 向上会经过其所有出现的后缀。唯一变化的是复杂度分析。



AC 自动机——复杂度分析

AC 自动机——构造

按长度从小到大考虑每个前缀（Trie 树上 BFS）。对于一个前缀，从 Trie 树的父亲节点开始向上跳 fail（不包含自己），直到某个前缀能够扩展出 S_i（在 Trie 树上有这个儿子）。

KMP——复杂度证明

记势能为当前点的长度。每次向上跳时势能减少，只在扩展时增加势能。因此复杂度为 $O(n)$。

在树上这不一定对：可能一个长度很大的点有很多儿子，每一个都向上跳回去。不过可以证明，沿着 Trie 中的每个字符串走过去，这条路径上的代价总和是 $O(|S_i|)$。因此总复杂度是 $O(\sum |S|)$。

由此，如果是输入若干个字符串，然后我们建 Trie。那直接这样做复杂度就是对的。但不一定是这种情况……

AC 自动机——构造（2）

另一种情形：给一个 Trie，对这个 Trie 建 AC 自动机。此时 $\sum|S|$ 不一定只有 Trie 大小级别。此时就不能用之前的算法了。

AC 自动机——构造

按长度从小到大考虑每个前缀（Trie 树上 BFS）。对于一个前缀，从 Trie 树的父亲节点开始向上跳 fail（不包含自己），直到某个前缀能够扩展出 S_i（在 Trie 树上有这个儿子）。

现在的复杂度在于向上跳 fail 找一个字符的扩展时，可能需要在 fail 树上一个地方重复跳多次。为避免重复，考虑把这个扩展的结果存下来，就是之前提到的转移边：

- 转移边：对于每个前缀，$next_c$ 表示向这个前缀加入字符 c 后，其最长的匹配某个前缀的后缀。

AC 自动机——构造（2）

- 状态为这些串的所有前缀。
- 对于每个节点，fail 指向其最长的匹配某个前缀的后缀。
- 转移边：对于每个节点，$next_c$ 表示向这个前缀加入字符 c 后，其最长的匹配某个前缀的后缀。

如何求转移边?

- 如果 Trie 树上它就有这个儿子，那么 $next_c$ 就指向这个儿子。
- 否则，按照之前的做法，我们沿着 fail 向上跳，直到有一个点有这个儿子。但如果我们求出了 $fail_u$ 的转移边，那跳过去之后我们就直接知道了答案。

Trie 树上 BFS 即可，复杂度 $O(n|\Sigma|)$。字符集过大时不建议使用。

AC 自动机——多串匹配

AC 自动机

给定多个串 $S_1, S_2, \ldots$，我们构造：

- 状态为这些串的所有前缀。
- 对于每个节点，fail 指向其最长的匹配某个前缀的后缀。
- 转移边：对于每个节点，$next_c$ 表示向这个前缀加入字符 c 后，其最长的匹配某个前缀的后缀。

类似 KMP 的应用方式，考虑读入一个 T，我们看它匹配了哪些串。

AC 自动机

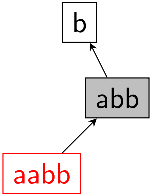
给定多个串 $S_1, S_2, \ldots$，我们构造：

- 状态为这些串的所有前缀。
- 对于每个节点，fail 指向其最长的匹配某个前缀的后缀。
- 转移边：对于每个节点，$next_c$ 表示向这个前缀加入字符 c 后，其最长的匹配某个前缀的后缀。

类似 KMP 的应用方式，考虑读入一个 T，我们看它匹配了哪些串。和之前一样，每读入一个字符，我们更新当前 T 的后缀最长可以和哪个前缀匹配，然后通过这个看是否有匹配……

与 KMP 不同的是，只有一个串的时候，匹配当且仅当当前最长匹配到整串。但多个串的时候，有可能现在匹配了这个串，但同时匹配了别的串一个更长的前缀，这时就会记录到那个前缀上。

与 KMP 不同的是，只有一个串的时候，匹配当且仅当当前最长匹配到整串。但多个串的时候，有可能现在匹配了这个串，但同时匹配了别的串一个更长的前缀，这时就会记录到那个前缀上。



因此，每匹配一位，我们需要在 fail 树（不是 Trie 树）上，把当前点到根上的匹配次数都 $+1$。这样才能正确算出每个给定串的出现次数。

这也相当于每个串的匹配次数是有多少次匹配停在它 fail 树子树内。那么在线询问可以 DFS 序+区间求和，离线可以最后一次树上子树和。

AC 自动机——应用

AC 自动机——多串匹配

对所有 S_i 建 AC 自动机，然后对 T 做匹配。一个 S_i 出现的次数等于匹配时有多少步停在了它 fail 树子树内。

AC 自动机——多串匹配

对所有 S_i 建 AC 自动机，然后对 T 做匹配。一个 S_i 出现的次数等于匹配时有多少步停在了它 fail 树子树内。

有一些直接的板子，形如：

单词

给一些串，问每个串在所有串中出现了多少次。

AC 自动机——应用

AC 自动机——多串匹配

对所有 S_i 建 AC 自动机，然后对 T 做匹配。一个 S_i 出现的次数等于匹配时有多少步停在了它 fail 树子树内。

阿狸的打字机

给一棵 Trie（不保证总串长很小），每次给出 Trie 上两个点，问一个点对应的字符串在另一个点对应的字符串中出现了多少次。

AC 自动机——应用

AC 自动机——多串匹配

对所有 S_i 建 AC 自动机，然后对 T 做匹配。一个 S_i 出现的次数等于匹配时有多少步停在了它 fail 树子树内。

Divljak

给 n 个串 S_i。每次操作：

- 加入一个询问串 T。
- 询问某个 S_i 在多少个之前加入的 T 中出现过。

AC 自动机——应用

AC 自动机——多串匹配

对所有 S_i 建 AC 自动机，然后对 T 做匹配。一个 S_i 出现的次数等于匹配时有多少步停在了它 fail 树子树内。

还有一类题目——问 T 是否不包含任意一个 S_i。

文本生成器

给一些串 S_i，询问有多少长度为 m 的串，使得至少有一个 S_i 在里面出现了。
$m, \sum |S_i| \le 1000$

然后还有 带一点点限制的计数路径数，10^9 的计数路径数。

目录

现在你有一个字符串 S，然后读入一个字符串 T，你需要判断 T 是不是 S 的子序列。

相当于，我们需要在 S 中选出一些递增的位置，使得它们拼起来等于 T。那贪心地想，我们应该让第一个位置尽量靠前，然后在第二个尽量靠前，接下来也一样。如果能匹配，那这样一定能匹配到。

因此判定方法是，记录匹配 T 之前的部分需要选到的位置 now，加入下一个字符后，选择 now 之后这个字符下次在 S 出现的位置。

子序列自动机

现在你有一个字符串 S，然后读入一个字符串 T，你需要判断 T 是不是 S 的子序列。

相当于，我们需要在 S 中选出一些递增的位置，使得它们拼起来等于 T。那贪心地想，我们应该让第一个位置尽量靠前，然后在第二个尽量靠前，接下来也一样。如果能匹配，那这样一定能匹配到。

因此判定方法是，记录匹配 T 之前的部分需要选到的位置 now，加入下一个字符后，选择 now 之后这个字符下次在 S 出现的位置。这可以写成自动机的形式，now 即为状态：

子序列自动机

- 状态是 S 的每个下标，表示当前贪心匹配到了这个位置。
- 转移 $next_{i,c}$ 表示位置 i 之后字符 c 第一次出现的位置。

根据刚才的分析，走 next 能走出的所有串正是 S 的所有子序列。

自动机小练习

子序列自动机

- 状态是 S 的每个下标，表示当前贪心匹配到了这个位置。
- 转移 $next_{i,c}$ 表示位置 i 之后字符 c 第一次出现的位置。

所有从起点出发，沿着 next 能走到的串是 S 的所有子序列。

还有一些问题也可以表示为这样的自动机：例如，之前提到的“不匹配任意一个 S_i 的字符串”[3]，以及之后将会看到的“S 的所有子串”。

有这样一类问题：问 S 的所有子序列/子串/……中的一些性质（数量，字典序第 k 小，……）。因为所有串正是自动机上从起点走出的所有路径，我们能利用自动机的性质来解决问题。

[3]与另外两个不同的是，它的转移是有环的，一般会限制长度。限制长度不超过 r 时可以看成把长度也加入状态，即复制 r 次。

自动机小练习——计数

自动机

- 有若干状态，每个状态 u 有转移 $next_{i,c}$，表示向后加入字符 c 后转移到的状态（可以不存在）。
- 所有从起点出发，沿着 next 能走到的串即为自动机接受的所有串，它们正好是满足某个性质的所有串。

求自动机接受（即满足性质）的串数量。例如子序列计数。

自动机小练习——DP

自动机

- 有若干状态，每个状态 u 有转移 $next_{i,c}$，表示向后加入字符 c 后转移到的状态（可以不存在）。
- 所有从起点出发，沿着 next 能走到的串即为自动机接受的所有串，它们正好是满足某个性质的所有串。

求最短的不被接受的串。

自动机小练习——DP

求最短的不被接受的串。

最短不公共子序列

给定 A, B，求最短的串，使得它是 A 的子序列但不是 B 的子序列。$n \leq 2000$

自动机小练习——字典序

自动机

- 有若干状态，每个状态 u 有转移 $next_{i,c}$，表示向后加入字符 c 后转移到的状态（可以不存在）。
- 所有从起点出发，沿着 next 能走到的串即为自动机接受的所有串，它们正好是满足某个性质的所有串。

按字典序从小到大枚举所有自动机接受的串。

自动机小练习——字典序

自动机

- 有若干状态，每个状态 u 有转移 $next_{i,c}$，表示向后加入字符 c 后转移到的状态（可以不存在）。
- 所有从起点出发，沿着 next 能走到的串即为自动机接受的所有串，它们正好是满足某个性质的所有串。

求字典序第 k 小的接受串。

自动机小练习（2）

求字典序第 k 小的接受串，但是多组询问（可以认为每次只输出答案的最后一位）

有一个基于轻重链剖分的思想，也被称为 DAG 剖分。

目录

Manacher

在一些情况下，我们想要维护一个串 S 中所有回文子串的信息。

回文串的一个性质：两侧各自删去一个字符还是回文串。因此每个回文串都对应一个中心，以及从中心延展的回文长度。那么我们只需要

对于每个位置，求出以这个位置为中心的最长回文子串长度。

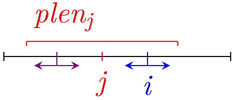
注意从中心角度看，实际上有两种回文串：以字符为中心（奇数长度）和以两个字符中间为中心（偶数长度）。一种偷懒方式是把字符串 abca 变成 a_b_c_a。

这当然可以 Hash 然后二分。但有一个比两个方向 Hash 还要简单的做法。

Manacher

对于每个位置，求出以这个位置为中心的最长回文子串长度 $plen_i$。

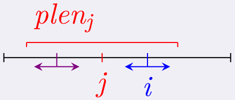
从小到大依次求出每个位置的答案。对于位置 i，考虑找到之前的一个位置 j，使得 j 为中心扩展的回文串覆盖到了 i。那么借助 j 的对称性：



从 i 出发向两侧匹配的过程和从对称位置 $2j-i$ 出发的过程完全一致——直到超出 j 给的对称性之前，那么借用 $2j-i$ 扩展的长度，

- 如果 i 加上这个长度小于 j 覆盖的 $j+plen_j$，那么不能匹配导致终止的这一步也对称过来了，从而完全一致地，$plen_i = plen_{2j-i}$。
- 否则，超出 j 覆盖范围的部分无法确认，因此我们继续暴力扩展。

Manacher's Algorithm



- 如果 i 加上这个长度小于 j 覆盖的 $j+plen_j$，那么不能匹配导致终止的这一步也对称过来了，从而完全一致地，$plen_i = plen_{2j-i}$。
- 否则，超出 j 覆盖范围的部分无法确认，因此我们继续暴力扩展。

为了最好利用之前的回文，显然选择能覆盖到最靠右的 j 最好，也就是 $j+plen_j$ 最大。

然后这就对了：从覆盖最远的位置开始继续扩展，那么每一步都会让最大 $i+plen_i$ 覆盖更远。复杂度 $O(n)$。

Manacher

Manacher's Algorithm

对于每个位置，求出以这个位置为中心的最长回文子串长度 $plen_i$。

这样，我们就把 S 中所有出现的回文串（不去重）表示成了如下 n 组：以 i 为中心，向两侧扩展不超过 $plen_i$ 长度。

简单小练习：

最长双回文串

求出 S 最长的子串，使得其可以被表示为两个非空回文串的拼接。

回文树

通过中心+扩展长度的表示，我们能找到所有回文串的所有出现。但这样并不能看出哪些串是相同的，或者说统计每个回文串出现了多少次。

那么，我们希望有一个结构，能够表示 S 所有的本质不同回文子串。

给定 S，我们构造一个数据结构，其中

- 每个节点对应 S 的一个回文子串。

接下来，我们考虑如何来构造这一结构，以及我们需要维护什么样的信息。

给定 S，我们构造一个数据结构，其中

- 每个节点对应 S 的一个回文子串。

考虑增量构造：每次向后加入一个字符，然后更新新出现的回文子串。那么记当前处理的前缀为 $S[1:i]$，在加入第 $i+1$ 个字符后，会多出来哪些回文串？

加入的回文串一定形如 $S[j:i+1]$。但这样的回文串都对应了一个以 i 结尾的回文串 $S[j+1:i]$。因此，找出以 i 结尾的回文串，然后看哪些可以扩展到 $i+1$，我们就能知道加入 $i+1$ 可能新增的回文串。

回文树——fail

给定 S，我们构造一个数据结构，其中

- 每个节点对应 S 的一个回文子串。

考虑增量构造：每次向后加入一个字符，然后更新新出现的回文子串。那么记当前处理的前缀为 $S[1:i]$，在加入第 $i+1$ 个字符后，会多出来哪些回文串？

加入的回文串一定形如 $S[j:i+1]$。但这样的回文串都对应了一个以 i 结尾的回文串 $S[j+1:i]$。因此，找出以 i 结尾的回文串，然后看哪些可以扩展到 $i+1$，我们就能知道加入 $i+1$ 可能新增的回文串。

我们需要知道 $S[1:i]$ 的所有回文后缀。学习 KMP 的思路，我们定义：

- 对于每个节点，$fail_u$ 表示它的最长回文后缀。

类似地，我们记录当前整串的最长回文后缀，从这里沿着 fail 走可以得到所有的回文后缀。

回文树——fail

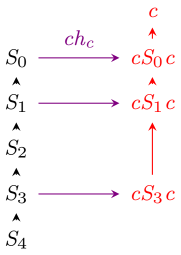
给定 S，我们构造一个数据结构，其中

- 每个节点对应 S 的一个回文子串。
- 对于每个节点，$fail_u$ 表示它的最长回文后缀。

维护当前的最长回文后缀。加入新字符后，我们可以像这样求出新加入字符涉及到的回文串 (链接表示 fail)：

$$\begin{array}{cc} & c \\ & \uparrow \\ S_0 & cS_0c \\ \uparrow & \uparrow \\ S_1 & cS_1c \\ \uparrow & \uparrow \\ S_2 & \mid \\ \uparrow & \mid \\ S_3 & cS_3c \\ \uparrow & \\ S_4 & \end{array}$$

回文树——ch



此时右边是加入新字符后所有的后缀回文串。但还需要去重：我们需要判断每一个扩展出来的 cS_ic 是新出现的回文串，还是之前就有的。

为了去重，我们定义 $ch_{u,c}$ 表示向 u 对应字符串两侧扩展字符 c 后，在回文串上对应的字符串。那么只需要看是否之前就存在 $ch_{S_i,c}$。更新 ch 只需要加入新节点 cS_ic 的时候，把 $ch_{S_i,c}$ 连过去。

回文树——构造

给定 S，我们构造一个数据结构，其中

- 每个节点对应 S 的一个回文子串。
- 对于每个节点，$fail_u$ 表示它的最长回文后缀，然后
- $ch_{u,c}$ 表示向 u 对应字符串两侧扩展字符 c 后，在回文串上对应的字符串。

刚才的分析已经给出了一个构造方式：沿着 fail 向上找到当前所有回文后缀，看每一个能否向后扩展；对可以扩展的，用 ch 看是否需要加新的点；然后把 ch 连到扩展后的部分，再在扩展后的部分之间连 fail（它们是所有新的回文后缀）。

回文树——构造

给定 S，我们构造一个数据结构，其中

- 每个节点对应 S 的一个回文子串。
- 对于每个节点，$fail_u$ 表示它的最长回文后缀，然后
- $ch_{u,c}$ 表示向 u 对应字符串两侧扩展字符 c 后，在回文串上对应的字符串。

刚才的分析已经给出了一个构造方式：沿着 fail 向上找到当前所有回文后缀，看每一个能否向后扩展；对可以扩展的，用 ch 看是否需要加新的点；然后把 ch 连到扩展后的部分，再在扩展后的部分之间连 fail（它们是所有新的回文后缀）。

直接这样看好像复杂度不一定对。但回文串保证了一个关键性质：

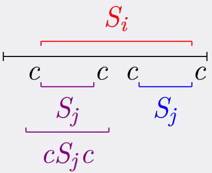
记向上跳到的第一个能扩展的回文后缀为 S_i，扩展后为 cS_ic。则对于之后所有可以扩展的后缀 S_j，它们的扩展 cS_jc 一定之前就已经存在。

回文树——构造

结论

记向上跳到的第一个能扩展的回文后缀为 S_i，扩展后为 cS_ic。则对于之后所有可以扩展的后缀 S_j，它们的扩展 cS_jc 一定之前就已经存在。

证明



回文树——构造

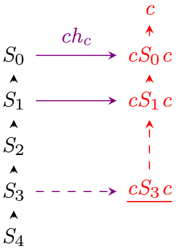
结论

记向上跳到的第一个能扩展的回文后缀为 S_i，扩展后为 cS_ic。则对于之后所有可以扩展的后缀 S_j，它们的扩展 cS_jc 一定之前就已经存在。

换言之，只需要修改标注的两条线，cS_1c 向上都不需要再改。

$$
\begin{tikzcd}
 & c \\
S_0 \arrow[r, "ch_c"] & cS_0c \arrow[u] \\
S_1 \arrow[u] \arrow[r] & cS_1c \arrow[u] \\
S_2 \arrow[u] & \\
S_3 \arrow[u] \arrow[r, dashed] & \underline{cS_3c} \arrow[uu, dashed] \\
S_4 \arrow[u] &
\end{tikzcd}
$$

回文树——构造



更新时，向上跳到第一个可以扩展的 S_i，此时

- 如果 cS_ic（$ch_{S_i,c}$）存在，那么上面的部分一定之前构造过了，只需要更新最长回文后缀。
- 否则，再向上找到第二个可以扩展的 S_j，此时 cS_jc 必然存在，并且上面都构造好了。只需要把新建 cS_ic 的 fail 更新为 cS_jc。同时更新 $ch_{S_i,c}$。

回文树——构造

回文树——构造

从最长回文后缀开始，向上跳到第一个可以扩展的 S_i，此时

- 如果 cS_ic（$ch_{S_i,c}$）存在，那么上面的部分一定之前构造过了，只需要更新最长回文后缀。
- 否则，再向上找到第二个可以扩展的 S_j，此时 cS_jc 必然存在，并且上面都构造好了。只需要把新建 cS_ic 的 fail 更新为 cS_jc。同时更新 $ch_{S_i,c}$。

需要判断一个串是否能扩展。但此时找到的 S_i 都是当前的后缀，因此只需要记录每个点的长度 len 即可判断。

回文树——构造

回文树——构造

从最长回文后缀开始，向上跳到第一个可以扩展的 S_i，此时

- 如果 cS_ic（$ch_{S_i,c}$）存在，那么上面的部分一定之前构造过了，只需要更新最长回文后缀。
- 否则，再向上找到第二个可以扩展的 S_j，此时 cS_jc 必然存在，并且上面都构造好了。只需要把新建 cS_ic 的 fail 更新为 cS_jc。同时更新 $ch_{S_i,c}$。

需要判断一个串是否能扩展。但此时找到的 S_i 都是当前的后缀，因此只需要记录每个点的长度 len 即可判断。

但还有一个小问题——假设现在往上跳到了空串，那么空串也可以扩展回文串。但空串向两侧扩展字符 c，得到 c 还是 cc?

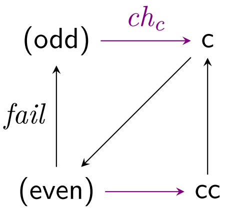
事实上两个应该被考虑。因此我们定义两个奇偶节点，分别考虑这两种情况。

回文树——奇偶节点

定义两个表示空串的节点——奇节点与偶节点。唯一的区别是，奇节点的 ch_c 指向字符串 c，而偶节点的 ch_c 指向字符串 cc。

巧妙的实现方式是，结合用 len 判是否扩展（$S_i = S_{i-len_u-1}$），然后偶节点 $len = 0$，奇节点 $len = -1$。

我们希望向上跳 fail 的时候这两种转移都要考虑，可以实现成如下方式：指向空串的 fail 都指向偶节点，然后偶节点 fail 指向奇节点。



回文树——参考实现

```
void init(){fail[0]=fail[1]=1;len[1]=-1;las=1;ct=1;}
void insert(int s,int c)
{
  while(st[s]!=st[s-len[las]-1])las=fail[las];
  if(ch[las][c])las=ch[las][c];
  else
  {
    int s1=++ct,tp=fail[las];
    while(st[s]!=st[s-len[tp]-1])tp=fail[tp];
    fail[s1]=ch[tp][c];ch[las][c]=s1;len[s1]=len[las]+2;las=ct;
  }
}
```

回文树——基本性质

有一些性质：

结论 1

任意串最多只有 n 个本质不同的回文子串。

证明：在上述构造中，每次加一个字符我们最多加一个点。

结论 2

上述构造的复杂度是 $O(n)$。

回文树——复杂度证明

线性复杂度证明

每次更新的过程是，从最长回文后缀开始往上跳 fail 直到可以扩展。如果那里 ch_c 已经存在了就直接接过去，否则加一个点，然后向上跳到第二个可以扩展的点来更新 fail。

进行势能分析。我们定义势能 ϕ 为当前记录的最长回文后缀节点在 fail 树上的高度。此时，

- 向上跳 fail 的步骤，每次跳 ϕ 减一，可以均摊分析。
- 然后考虑跳 ch_c 的一步。沿着 fail 向上跳是遍历当前的所有回文后缀。那么根据之前的分析，加一个 c 之后的回文后缀都是先前的回文后缀——最多多一个 1 或 2 字符的。

$$\begin{tikzcd}
 & cS_3c \arrow[rr] & & cS_1c \arrow[r] & cS_0c \arrow[r] & c \\
S_4 \arrow[r] & S_3 \arrow[u] \arrow[r] & S_2 \arrow[r] & S_1 \arrow[r] & S_0 &
\end{tikzcd}$$

回文树——复杂度证明

线性复杂度证明 Part 2

$$
\begin{tikzcd}
& cS_3c \arrow[rr] & & cS_1c \arrow[r] & cS_0c \arrow[r] & c \\
S_4 \arrow[r] & S_3 \arrow[u] \arrow[r] & S_2 \arrow[r] & S_1 \arrow[u] \arrow[r] & S_0 &
\end{tikzcd}
$$

最后一步是，如果我们加了新点，那还需要沿着原来的 fail 向上找到第二个能扩展的，然后把 fail 接过去。

此时每跳一步，就会说明 ϕ 又减了 1：这部分跳过的在新 fail 上都没有了。

因此除去 $O(1)$ 步外，剩下的步骤都被势能抵消了，从而均摊复杂度 $O(n)$。

回文树——回文自动机

回文树

给定 S，我们构造一个数据结构，每个节点对应 S 的一个回文子串。对于每个点，我们维护

- len_u 表示这个点对应串的长度，然后
- $fail_u$ 表示它的最长回文后缀，以及
- $ch_{u,c}$ 表示向两侧加入 c 后对应的串（如果那还是 S 的子串）

这样，我们就维护了 S 所有本质不同回文子串之间的一个结构。fail 表示了这些子串之间的一个后缀关系。ch 表示了向外扩展的关系。

不简单的练习题：Virus Synthesis。

回文树——回文自动机

回文树

给定 S，我们构造一个数据结构，每个节点对应 S 的一个回文子串。对于每个点，我们维护

- len_u 表示这个点对应串的长度，然后
- $fail_u$ 表示它的最长回文后缀，以及
- $ch_{u,c}$ 表示向两侧加入 c 后对应的串（如果那还是 S 的子串）

沿着 ch 走，可以支持读入一个回文串，判断它是不是 S 的子串。因此这也被称作回文自动机。

注意这个和一般自动机读字符的顺序不一样。

回文树——出现位置/Occ

Occ 是出现位置的简写。

回文树直接给出了 S 中所有的本质不同子串。但在处理 S 中问题时，我们不仅希望知道这些串，还希望把它们和 S 关联起来——或者说，求出每个回文串在 S 中出现在了哪里。

$Occ(T)$ 表示 T 在 S 中出现的位置构成的集合。视情况不同，我们可能使用出现的开头或结尾位置表示它。

怎么知道 Occ 呢?

Occ 是出现位置的简写。

回文树直接给出了 S 中所有的本质不同子串。但在处理 S 中问题时，我们不仅希望知道这些串，还希望把它们和 S 关联起来——或者说，求出每个回文串在 S 中出现在了哪里。

$Occ(T)$ 表示 T 在 S 中出现的位置构成的集合。视情况不同，我们可能使用出现的开头或结尾位置表示它。

怎么知道 Occ 呢？ 我们在构造的时候，每一步维护了当前前缀的最长回文后缀 las_i，并且说，沿着 fail 跳就可以得到它的所有回文后缀。

注意到，出现次数等于“它是多少个后缀的前缀”。那么从每一个 las_i 开始沿着 fail 树向上跳，将经过的点出现次数 $+1$，最后就得到了所有串的出现次数。换言之：

记 las_i 是 $S[1:i]$ 的最长回文后缀。回文树上一个点 u 对应串在 S 中的所有出现位置（结束位置）即为 fail 树上 u 子树内包含的所有 las_i 对应下标。

回文树——Occ

记 las_i 是 $S[1:i]$ 的最长回文后缀。回文树上一个点 u 对应串在 S 中的所有出现位置（结束位置）即为 fail 树上 u 子树内包含的所有 las_i 对应下标。

维护出现次数的板子：

回文串

给定 S，求其回文子串中最大的 $|T|\cdot|Occ(T)|$。

快乐的 JYY

给定 S,T，对所有回文子串 A 求和 $|Occ_S(A)|\cdot|Occ_T(A)|$。

回文树——Occ

记 las_i 是 $S[1:i]$ 的最长回文后缀。回文树上一个点 u 对应串在 S 中的所有出现位置（结束位置）即为 fail 树上 u 子树内包含的所有 las_i 对应下标。

还可以维护出每个点的所有出现位置，例如使用线段树合并。然后可以做很多出现位置相关的查询。

但好像很少有这样的题目，原因可能是大家更愿意出后缀树。那里的题目都可以复制到回文串的情况下，甚至这个也可以。

回文树——扩展

双向插入回文树

如题，现在还需要支持向前加入字符。

和下一章的情况不同，这实际上是一个脑筋急转弯。

目录

后缀数据结构

之前的一些算法，例如 KMP 和 AC 自动机，都是维护了一些串中所有前缀的信息。~~这就是为什么我写了个前缀数据结构在那，实际上应该没有这个名字。~~

如果我们能对一个串的所有后缀做类似的事，那我们就能维护一个串所有后缀的前缀信息，即所有的子串信息。因此后缀数据结构是一种解决子串问题的有效方案。

后缀数组

定义：将一个串的所有后缀按照字典序排序。

$\boxed{0}$

$\boxed{0\ 1\ 0\ 1\ 1\ 0}$

$\boxed{0\ 1\ 1\ 0}$

$\boxed{1\ 0}$

$\boxed{1\ 0\ 1\ 1\ 0}$

$\boxed{1\ 1\ 0}$

记号约定

- sa_i 表示第 i 小的后缀是谁。
- rk_i 表示第 i 个后缀的顺序。

后缀数组——倍增做法

~~二分+hash~~

经典的倍增求法：

- 在第 0 轮，求出每个后缀只保留至多前 1 个字符后的排序结果。
- 在第 i 轮，求出每个后缀只保留至多前 2^i 个字符的排序结果。

~~二分+hash~~

经典的倍增求法：

- 在第 0 轮，求出每个后缀只保留至多前 1 个字符后的排序结果。
- 在第 i 轮，求出每个后缀只保留至多前 2^i 个字符的排序结果。

对单个字符排序是简单的：桶排序。

```
for(int i=1;i<=n;i++)ct[i]=0;
for(int i=1;i<=n;i++)ct[s[i]]++;
for(int i=1;i<=n;i++)ct[i]+=ct[i-1];
for(int i=n;i>=1;i--)sa[ct[s[i]]--]=i;//rk[i]=ct[s[i]]--;
```

那第二轮，对两个字符排序呢？

后缀数组——基数排序

现在考虑对 n 个长度为 2 的字符串按照字典序排序，每个串记作 (a_i,b_i)。
需要的性质：按照 a_i 排序，a_i 相同时按照 b_i 排序……

```
for(int i=n;i>=1;i--)sa[ct[s[i]]--]=i;//rk[i]=ct[s[i]]--;
```

桶排序的性质：a_i 相同时，按照最后遍历顺序的倒序排序。从而我们可以

基数排序

首先按照 b_i 做一轮桶排序。然后按照 a_i 做桶排序，但在最后一步时按照上一轮结果**倒序**遍历。

- 在第 0 轮，求出每个后缀只保留至多前 1 个字符后的排序结果。
- 在第 i 轮，求出每个后缀只保留至多前 2^i 个字符的排序结果。

在第 i 轮，我们对每个后缀的前 2^i 个字符进行了排序。然后考虑第 $i+1$ 轮，对前 2^{i+1} 个字符排序……

$S[j:]$ 的前 2^{i+1} 个字符由两段 2^i 组成，其一是 $S[j:]$ 的前 2^i 个字符，其二是 $S[j+2^i:]$ 的前 2^i 个字符。因此，如果我们对每个长度为 2^i 的子串排序后得到了当前的 sa，那下一步只需要对所有 (rk_j, rk_{j+2^i})[4] 排序。用刚才的基数排序即可。

[4]如果超出边界，那后半是空串，字典序最小

后缀数组——去重

有一个细节问题：按照上面的方法倍增时，如果两个长度为 2^i 的串完全相同，则它们的 rk 也应该相同。但桶排序不满足这件事。

因此需要一个去重步骤：得到新的 sa/rk 后，依次看相邻两个串是否完全相同，相同就合并 rk。这就是判断两个 (rk_j, rk_{j+2^i}) 是否相同。

每一轮复杂度 $O(n)$，总复杂度 $O(n \log n)$。

后缀数组——去重

有一个细节问题：按照上面的方法倍增时，如果两个长度为 2^i 的串完全相同，则它们的 rk 也应该相同。但桶排序不满足这件事。

因此需要一个去重步骤：得到新的 sa/rk 后，依次看相邻两个串是否完全相同，相同就合并 rk。这就是判断两个 (rk_j, rk_{j+2^i}) 是否相同。

每一轮复杂度 $O(n)$，总复杂度 $O(n\log n)$。

常见实现细节：基数排序的第一步是按照 rk_{j+2^i} 基数排序，相当于将位置 j 按照 $j+2^i$ 向后长度 2^i 的子串排序。因此如果之前求 rk 的时候也记录了 sa，那么可以简化这一轮：

```
for(int i=n-l+1;i<=n;i++)b[++ct]=i;
for(int i=1;i<=n;i++)if(sa[i]>l)b[++ct]=sa[i]-l;
```

后缀数组——扩展

- 如果一开始字符集特大，可以先排序一次让字符集变回 $[n]$。
- 如果多个串一起做（常见 LCP 题目），简单做法是往中间加特殊分隔符。注意最好不要加相同分隔符。
- 上述问题还有更好的解决方法：事实上我们可以对一个 Trie 状物做后缀排序：给一个 Trie，每个点对应它出发向上到根的串，然后给这些排序。做法和之前的倍增一样——第 i 轮，我们对每个点向上的路径的前 2^i 位进行排序，然后两个 2^i 组合 2^{i+1}（模板）。
- 特例：如果是个环（例如字符加密），那么按照经典方法，复制两遍做即可考虑所有环后缀。
- 存在复杂度更好的算法，例如 SA-IS 与 DC3。如果你感兴趣可以自行学习，~~因为我不会~~。

将后缀排序给出了一些信息，但有些时候我们希望更多的信息。

例如，SA 可以直接比较两个后缀的字典序。但如果我们想要比较两个子串，此时直接比较后缀字典序的话，无法判断是否相等。

因此，我们希望记录相邻两个后缀比较时，比较到什么时候变得不同。据此，我们定义：

SA——height

height 表示 SA 中相邻两个后缀的公共前缀长度，即

$$height_i = LCP(S[sa_i :], S[sa_{i+1} :])$$

后缀数组——height 计算

SA——height

height 表示 SA 中相邻两个后缀的公共前缀长度，即

$$height_i = LCP(S[sa_i :], S[sa_{i+1} :])$$

~~二分+hash~~

结论

$$height_{rk_{i+1}} \geq height_{rk_i} - 1$$

人话：$S[i+1 :]$ 和其字典序下一个后缀的 LCP 长度最多比 $S[i :]$ 与其下一个的 LCP 长度少 1。

证明：不妨设排序后，$S[i :]$ 的下一个是 $S[j :]$，则 $S[j :]$ 字典序大于 $S[i :]$。如果这个 LCP 长度至少是 1，那么比较下一位，$S[j+1 :]$ 字典序也大于 $S[i+1 :]$，且这个 LCP 长度只比之前少 1。因此位置 $i+1$ 对应的 height 至多比位置 i 对应的 height 少 1。

SA——height

height 表示 SA 中相邻两个后缀的公共前缀长度，即

$$height_i = LCP(S[sa_i :], S[sa_{i+1} :])$$

结论

$$height_{rk_{i+1}} \geq height_{rk_i} - 1$$

人话：$S[i+1 :]$ 和其字典序下一个后缀的 LCP 长度最多比 $S[i :]$ 与其下一个的 LCP 长度少 1。

那么有如下线性做法：

依次求 $height_{rk_i}$，即后缀 i 和字典序下一个的 LCP。每次从上一个长度（$height_{rk_{i-1}}$）减一开始贪心向后匹配。

后缀数组——height 视角

将所有后缀字典序排序，求出相邻后缀的 LCP 长度作为 height。

基础小练习：

- 不重复地找到所有本质不同子串。

所有子串即为所有后缀的前缀。

后缀数组——height 视角

将所有后缀字典序排序，求出相邻后缀的 LCP 长度作为 height。

基础小练习：

- 不重复地找到所有本质不同子串。

所有子串即为所有后缀的前缀。现在按字典序加入每个后缀的前缀，考虑重复了多少。与字典序上一个后缀的 LCP 部分内都是重复的，但之后部分都比前面字典序大。因此这个后缀带来的本质不同子串正是其所有长度大于当前 height 的前缀。答案是 $\frac{1}{2}n(n+1)-\sum height_i$。

弦论 Task 1

求第 k 小本质不同子串。

后缀数组——height 视角

所有子串即为所有后缀的前缀。现在按字典序加入每个后缀的前缀，考虑重复了多少。与字典序上一个后缀的 LCP 部分内都是重复的，但之后部分都比前面字典序大。因此这个后缀带来的本质不同子串正是其所有长度大于当前 height 的前缀。

因此本质不同子串可以用 n 组 (s, l, r) 表示，其中每一组表示以 s 开头，在 $[l, r]$ 间结尾的所有子串。然后不仅可以做本质不同子串计数，还可以对本质不同子串求和一些别的东西 (例：每种字符有个权值，求总和非负的本质不同子串数量)

后缀数组——LCP

将所有后缀字典序排序，求出相邻后缀的 LCP 长度作为 height。

那任意两个后缀的 LCP 呢？ 因为是字典序排序，有如下结论：

将后缀按字典序排序后，两串 LCP 即为它们中间所有的 height 最小值。

证明：因为 LCP 可以传递（$LCP(a,b) \geq \min(LCP(a,c), LCP(b,c))$），所以 LCP 至少是这个值。又因为是字典序，所以如果相邻两串某个字符不同，那么跨越这里的两个串要么之前就有不同的，要么之前相同，但这个字符必然不同。

这样就把 LCP 变成了区间 $\min$，一个序列上的问题。

后缀数组——LCP

将所有后缀字典序排序，求出相邻后缀的 LCP 长度作为 height。
将后缀按字典序排序后，两串 LCP 即为它们中间所有的 height 最小值。

很有用的工具：使用 ST 表预处理即可 $O(1)$ 算两后缀的 LCP。

另一个很有用的工具：比较两个子串的字典序。看一下两个后缀的字典序顺序，再用两个后缀的 LCP 判断是否相等。

后缀数组——LCP

将所有后缀字典序排序，求出相邻后缀的 LCP 长度作为 height。
将后缀按字典序排序后，两串 LCP 即为它们中间所有的 height 最小值。

- 求两串最长公共子串。

后缀数组——LCP

将所有后缀字典序排序，求出相邻后缀的 LCP 长度作为 height。
将后缀按字典序排序后，两串 LCP 即为它们中间所有的 height 最小值。

生成魔咒

求出每个前缀内的本质不同子串数。

后缀数组——LCP/区间 $\min$

将后缀按字典序排序后，两个后缀的 LCP 长度就是 height 的区间 $\min$。

LCP 是 height 的区间 $\min$。这样的序列问题我们有很多处理方式：

差异

求 $\sum_{i<j} LCP(S[i:], S[j:])$.

品酒大会

对每个 i，求有多少对后缀的 LCP 至少是 i。

后缀数组——LCP/区间 $\min$

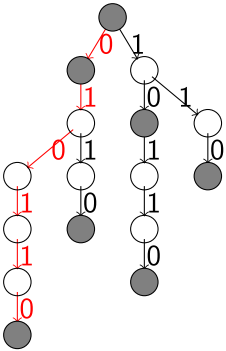
将后缀按字典序排序后，两个后缀的 LCP 长度就是 height 的区间 $\min$。

字符串

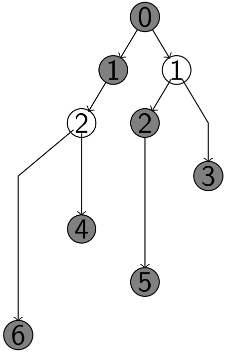
给一个串 S，多次询问，每次问 $S[a:b]$ 中任选一个子串，它和 $S[c:d]$ 的 LCP 的最大值。

后缀树

Trie 里面记录了所有输入 Trie 的前缀。为了维护子串信息，我们将所有后缀放入 Trie……



Suffix Trie



Suffix Tree

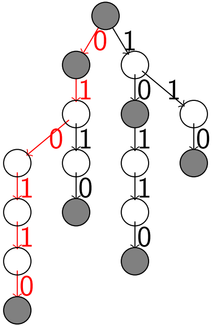
为了压缩，我们只需要保留后缀节点构成的虚树，以及虚树上每个节点的串长，记作 len_u。

后缀 Trie——更新

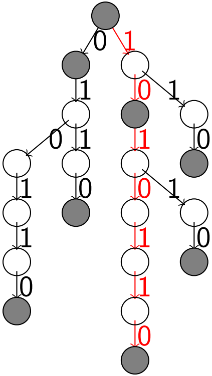
如何构造？ 考虑依次插入字符。加入字符后，我们需要更新后缀的 Trie。

向后插入？ 所有后缀都会变化，不太可行。

因此我们向前插入，加入一个后缀。



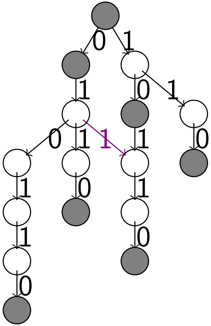
Before: S



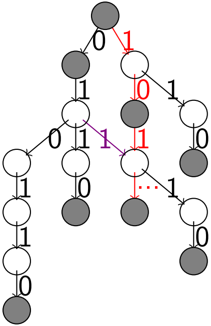
Insert 1: cS

为此，我们需要在 Trie 上定位新加入的后缀……这需要一些工具。

如果我们知道 $front_{u,c}$，表示节点 u **向前**加入字符 c 后所对应的后缀树节点，那就很简单了。



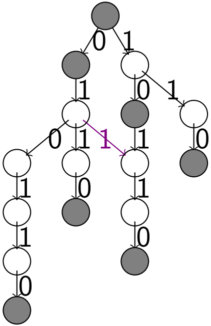
Using $front_1$

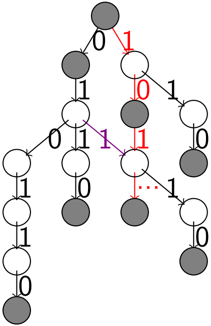


After: cS

后缀 Trie——更新 Trie

$front_{u,c}$ 表示节点 u **向前**加入字符 c 后所对应的后缀树节点。





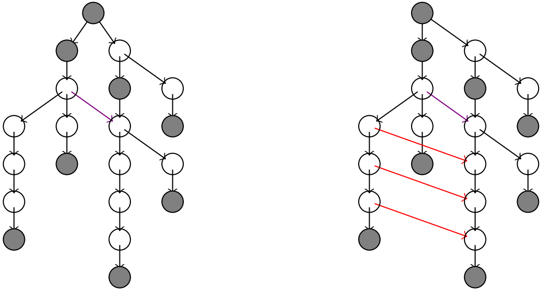
后缀树——更新 Trie

从整串 (S) 节点开始，向上找到第一个 $front_c$ 存在的节点，从指向的 $front_c$ 向下建新的链。

后缀 Trie——更新 front

$front_{u,c}$ 表示节点 u **向前**加入字符 c 后所对应的后缀树节点。

然后需要根据新加入的点更新 $front$。新加入的子串是 cS 的一些前缀。所以能 $front_c$ 转移到它的……正是 S 的一些前缀，向上找经过的那些。

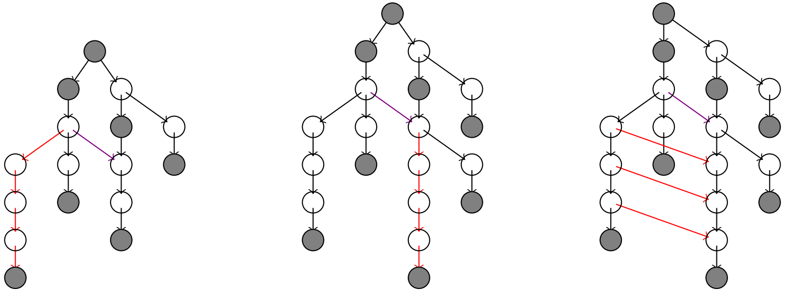


由于被 latex 击败，这里省略了最后一条 front

后缀 Trie——更新

因此更新只需要如下几步：记录 S 对应的节点，加入字符 c 时：

- 从 S 沿着树边向上，直到第一个 $front_c$ 存在的节点（或回到根）停止。
- 从该节点对应的 $front_c$ 开始向下建一条链，对应新加入的串。
- 将第一步向上走过部分的 $front_c$ 对应连向新加入部分。



Search　　Insert　　Update front

后缀 Trie

后缀 Trie

将 S 所有后缀加入 Trie，每个节点表示一个后缀的前缀，即 S 的一个子串。对于每个节点：

- fa_u 表示其在 Trie 树上的父亲节点。
- $front_{u,c}$ 表示其**向前**加入字符 c 后对应的节点。

后缀 Trie——构造

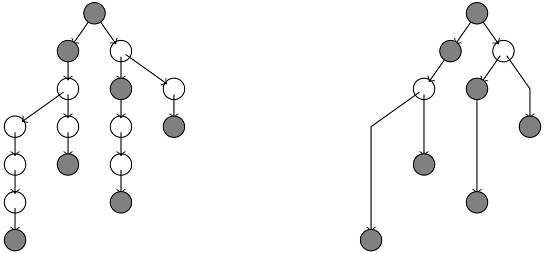
记录 S 对应的节点，加入字符 c 时：

- 从 S 沿着树边向上，直到第一个 $front_c$ 存在的节点（或回到根）停止。
- （更新 Trie）从该节点对应的 $front_c$ 开始向下建一条链，对应新加入的串。
- （更新 front）将第一步向上走过部分的 $front_c$ 对应连向新加入部分。

现在，把这些东西放到真正的后缀树上。

后缀 Trie → 树

真正的后缀树是后缀 Trie 的虚树。因此我们可以考虑使用上面的更新方式。唯一的区别是，一些链被压缩了起来。



压缩有什么影响？ 首先考虑压缩 $front$。好消息，没有影响：

一个压缩节点中，对于每一个 c，其中所有指向的 $front_c$ 在同一个压缩节点内。

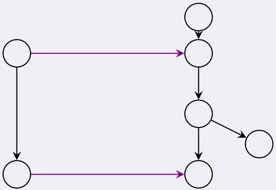
后缀 Trie $\to$ 树

结论

一个压缩节点中，对于每一个 c，其中所有指向的 $front_c$ 在同一个压缩节点内。

证明

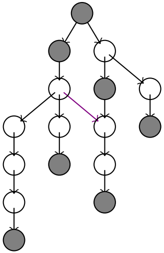
如果矛盾，一定出现了下图中的情况：



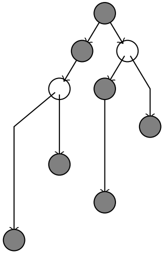
但这样的话，左边对应位置分叉的串也应该存在——右侧是左侧在前缀加一个字符的结果。

后缀树——更新

还有什么步骤中，压缩会导致问题？



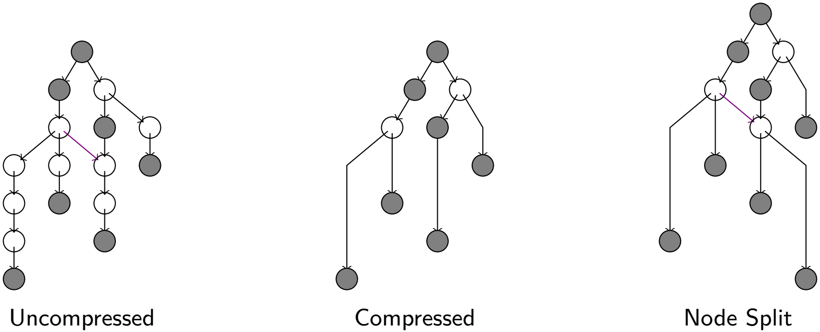
Uncompressed



Compressed

后缀树——更新

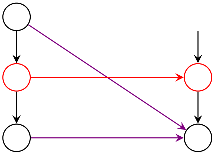
还有什么步骤中，压缩会导致问题？



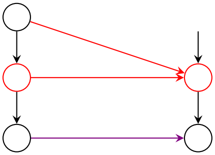
唯一的问题：可能找到的 $front$ 指向压缩节点的中间。此时应当拆点。

后缀树——拆点

那么拆点，会发生什么?



Before



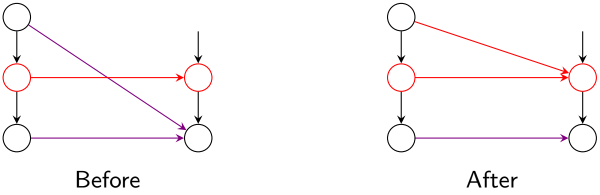
After

可能有一些连向被拆点的 front 转移。拆点之后我们需要把这其中的一些改为指向新的点。那么，哪些 front 应该指向拆出来的点?

[5]理论上应当按照 len 分，但此时 len 更大的先前都没有这条 front。

后缀树——拆点

那么拆点，会发生什么？



可能有一些连向被拆点的 front 转移。拆点之后我们需要把这其中的一些改为指向新的点。那么，哪些 front 应该指向拆出来的点？

拆出来的点是 cS 到根上的一段路径，那么里面的串是 cS 的前缀。那么 front 指向这里的一定是 S 的前缀。那是哪些前缀呢？ 正是那些指向被拆的点的。[5]

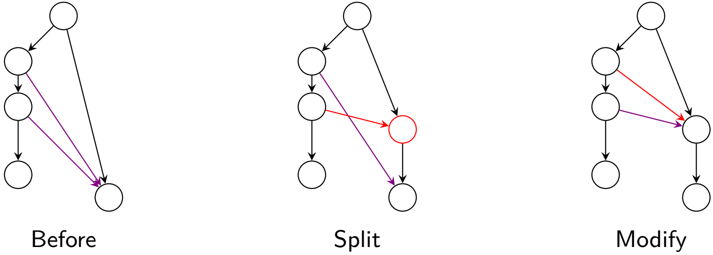
沿着左侧继续向上找到所有 $front_c$ 指向被拆的点的转移，全部换到新点上。

[5]理论上应当按照 len 分，但此时 len 更大的先前都没有这条 front。

沿着左侧继续向上找到所有 $front_c$ 指向被拆的点的转移，全部换到新点上。

按照 len 考虑，如果往上走到了一个 $front_c$ 指向更小的串的位置，那么之后 $front_c$ 就不会再指回来了。因此不断向上更新，到不能更新时停止即可。

另一个图示：



后缀树——代码示例

```
int ch[N][26],len[N],fa[N],cnt;
int insert(int las,int c)
{
  int u=++ct;
  len[u]=len[las]+1;
  while(las&&!ch[las][c])ch[las][c]=u,las=fa[las];
  int v=ch[las][c];
  if(!las)fa[u]=1;
  else if(len[v]==len[las]+1)fa[u]=v;
  else {
    int cl=++ct;
    len[cl]=len[las]+1;fa[cl]=fa[v];
    for(int i=0;i<26;i++)ch[cl][i]=ch[v][i];
    fa[v]=fa[u]=cl;
    while(las&&ch[las][c]==v)ch[las][c]=cl,las=fa[las];
  }
  return u;
}
```

有如下结论[6]：

- 拆点次数不超过 n。
- 更新 $front$ 的总次数为 $O(n)$。
- 最后有效的 $front$ 数量为 $O(n)$。

常见字符集的情况下，经典实现是每次复制整个 $front$ 数组。前两个结论可以说明复杂度是 $O(n|\Sigma|)$。

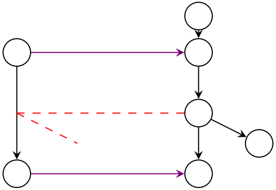
如果字符集很大，可以 map 实现转移。结合最后一个结论可以说明直接复制 map 复杂度也是 $O(n\log|\Sigma|)$。

[6]两处常数应该都不超过 3。

后缀树——更新次数证明

进行势能分析。定义势能 ϕ 为当前 S 对应节点在后缀树上的深度（压缩后）。分步考虑，

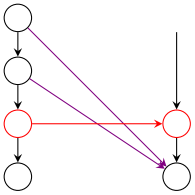
- 对于向上找 $front$ 的过程，每一步操作 ϕ 减一。
- 如果不需要拆点，可以证明跳到 $front$ 后 ϕ 最多加一：与之前的分析类似，沿 cS 向上遇到到每一个分叉在沿 S 向上时必定存在。这是因为，分叉对应字符串去掉开头的 c 就变成了 S 那边的情况。



后缀树——更新次数证明

最后是拆点后向上更新的情况。这段过程中，我们沿着 S 向上继续修改，原因是它们都指向了另一边压缩后的节点。

那么此时每向上一步，都会使得跳过去后深度减一——对应的点被压缩掉了。

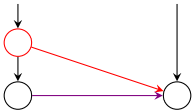


因此，除去跳 $front$ 的一步外，ϕ 加上总代价不会增加。跳的一步只影响常数。从而均摊复杂度为 $O(n)$。

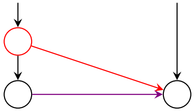
现在我们证明最后有效 front 只有 $O(n)$ 个。由于拆点只会增加转移数量，且前面说明了其它更新只有 $O(n)$ 次，这样可以说明直接复制的拆点方式实际上只复制了 $O(n)$ 个东西。

将转移分为两类：

- 没有经过压缩的（$len_v = len_u + 1$），这一类每个点只有一个入边，因此总共有 $O(n)$ 个。
- 经过压缩的，如下图：



此时，记红色节点对应串为 S，转移字符为 c。则这种情况说明：S 出现了很多次，使得它在后缀 Trie 上向下有分叉，但 cS 出现的次数更少，以至于没有分叉。



记红色节点对应串为 S，转移字符为 c。则这种情况说明：S 出现了很多次，使得它在后缀 Trie 上向下有分叉，但 cS 出现的次数更少，以至于没有分叉。

统计有多少转移 (S, cS) 满足 S 出现次数多于 cS 出现次数。考虑把整个串**倒过来**做，变成统计有多少 (S, Sc) 满足条件。

在倒过来的后缀 Trie 上，Sc 是 S 的儿子，即统计有多少儿子的子树大小小于父亲子树大小。答案即为虚树大小，因此也是 $O(n)$ 的[7]。

[7]有一个常数更小的证明。

后缀树——扩展

- 如果对多个串做，那么仍然可以分隔符。
- 也可以考虑不用分隔符：这里每一步是找到一个串 S，然后加一个新的后缀 cS。因此也可以加完一个串重新加新的串。但有一个问题：之前每一步一定会向下拉一条新的链，原因是这个后缀最长。但现在不一定有这种情况，可能跳完 front 就不需要向下加一条链了，需要特判。
- 一个更好的处理方法是倒过来建 Trie，然后在这上面 bfs 往前加字符。这样就保证了每次加入的一定是当前最长的。
- 但按照上述方式，给一个 Trie 建 SAM 的复杂度似乎不是 Trie 大小的，可能与总串长有关。需要注意。

后缀树——LCP

给定后缀树，询问两个后缀的 LCP。

因为这是个 Trie，LCP 就是两个后缀在 Trie 上的 LCA（的 len）。

后缀树——LCP

给定后缀树，询问两个后缀的 LCP。

因为这是个 Trie，LCP 就是两个后缀在 Trie 上的 LCA（的 len）。

回顾一下 SA：

- 后缀数组将所有后缀排序，然后求出了相邻两个后缀的公共前缀长度。
- 后缀树将所有后缀插入 Trie，然后求出了树的结构。

这两个结构是等价的（区间 $\min$ 上笛卡尔树 $\leftrightarrow$ LCA 用 DFS 序)。在后缀树上 DFS 即可求出后缀排序与 height。对 height 求笛卡尔树即可得到后缀树。

后缀树——LCP

因此用 SA/height 能做的题用后缀树的 Trie 结构也能做：

差异

求 $\sum_{i<j} LCP(S[i:], S[j:])$.

品酒大会

对每个 i，求有多少对后缀的 LCP 至少是 i。

后缀树相当于提供了另一个考虑 LCP 问题的视角。两种视角并不存在优劣之分，而是不同视角各自对一部分问题更直观。因此，实际问题需要灵活选择视角（例如，字符串 可能序列就比较直观。）

树上视角小练习：

字符串

给两个串 a, b。对每个串，取它所有长度为 k 的子串（不去重），记得到的可重集为 A, B。对 A, B 做匹配，每一对的分数是两串 LCP。求最大分数。

后缀树视角——Occ

给定后缀树上的一个节点[8]，其对应的串在 S 中出现了多少次?

这相当于，有多少个后缀包含它作为前缀。因此答案就是后缀树子树内的后缀节点数量。更一般地，

一个字符串在 S 中出现的所有起始位置[a]即为其子树中所有后缀的位置。

[a]常见实现会反过来建后缀树，此时相当于记录出现的所有结束位置。

维护这些位置，即可维护字符串在 S 中出现位置的信息。常见实现为启发式合并或（可持久化）线段树合并。

[8]常见情形是给出 l, r。由 l, r 定位子串的做法包括从 $S[l:]$ 开始倍增找到正确长度。

后缀树视角——Occ

一个字符串在 S 中出现的所有位置即为其子树中所有后缀的位置。

现在我们回顾一些经典操作：

求两串最长公共子串。

枚举本质不同子串。

更好的是，此时每一组对应一个节点，因此这一组中所有串的出现位置是相同的。因此可以做出现位置相关题目。

后缀树视角——Occ

所有本质不同子串按照节点分为多类，每一类表示为从某个位置开头，长度在 $[l, r]$ 之间。每一类内的字符串，出现位置相同，都是其子树中所有后缀的位置。

甲苯先生和大中锋的字符串

统计在 S 中出现 k 次的子串数量/长度。

弦论 Task 2

求字典序第 k 小子串，重复出现算多次。

后缀树视角——Occ

所有本质不同子串按照节点分为多类，每一类表示为从某个位置开头，长度在 $[l, r]$ 之间。每一类内的字符串，出现位置相同，都是其子树中所有后缀的位置。

一道题

给一个串 S，求有多少个子串满足：

- 在 S 中覆盖这个子串的每一次出现后，正好覆盖了 k 个位置。

后缀树视角——Occ 扩展

一个字符串在 S 中出现的所有位置即为其子树中所有后缀的位置。

区间本质不同子串个数

如题

后缀树——front

如果只用树边，那么它本质上还是和后缀数组一样的结构。你甚至可以直接拿 SA 建出后缀树的 Trie 结构。

但我们还有另一样东西——front。

后缀树 → 后缀自动机

回顾后缀树的结构：

- 每一个节点对应一些 S 的子串。具体来说，是从一个位置开始，长度在区间 $(height_{fa}, height_u]$ 间的那些串。
- 所有节点组成 Trie 树的结构，树上节点的父亲是它的前缀。
- 每个节点维护了 $front_c$，表示向前加入字符后字符串对应的节点。

现在 front 是向前加入，那如果我们反过来，对 S 的反串做后缀树呢？

后缀树 $\to$ 后缀自动机

回顾后缀树的结构：

- 每一个节点对应一些 S 的子串。具体来说，是从一个位置开始，长度在区间 $(height_{fa}, height_u]$ 间的那些串。
- 所有节点组成 Trie 树的结构，树上节点的父亲是它的前缀。
- 每个节点维护了 $front_c$，表示向前加入字符后字符串对应的节点。

现在 front 是向前加入，那如果我们反过来，对 S 的反串做后缀树呢？

- 每一个节点对应一些 S 的子串。现在是在某个位置结束，长度在一个区间内的串。
- 树上一个节点的父亲对应它的一个后缀。
- 每个节点维护 $next_c$，表示向后加入字符对应的节点。

后缀自动机

- 每一个节点对应一些 S 的子串。现在是在某个位置结束，长度在一个区间内的串。
- 树上一个节点的父亲对应它的一个后缀。
- 每个节点维护 $next_c$，表示向后加入字符对应的节点。

这只是把后缀树反过来，因此构造只需要沿用那个做法，但反过来插入字符。[9]

反过来之后，之前的东西仍然可以使用，但你需要反过来想：LCA 表示两个前缀的最长公共后缀。一个串的出现位置是它的子树内有多少前缀结点，然后这些对应了这个串的结束位置(因此它更经常被称为 endpos)。

那为什么要反过来呢？ 现在，next 变得更自然了。

[9] 因为后缀树是从后向前，因此后缀自动机反而是从前向后。
为什么经常会反着建后缀树呢？ 因为经典教学方式是直接讲后缀自动机，再说反串后缀自动机就是后缀树。但我认为后缀树是更自然的表示。

后缀自动机——next

- 每一个节点对应一些 S 的子串。
- 每个节点维护 $next_c$，表示向后加入字符对应的节点。

沿着 next 走相当于，我们不断向一个串 T 的结尾加入字符，然后询问 T 还是不是 S 的子串。因此，沿着 next 走出的所有路径正对应 S 的所有子串。这就是一个接受 S 所有子串的自动机。

后缀自动机

沿着 next 转移能走出的所有字符串即为 S 的子串。

现在，回顾 自动机小练习。全文抄写可得：

多次询问，每次给一个 T，问 T 是不是 S 的子串。

后缀自动机

沿着 next 转移能走出的所有字符串即为 S 的子串。

现在，回顾 自动机小练习。全文抄写可得：

多次询问，每次给一个 T，问 T 是不是 S 的子串。

本质不同子串计数。

后缀自动机

沿着 next 转移能走出的所有字符串即为 S 的子串。

现在，回顾 自动机小练习。全文抄写可得：

多次询问，每次给一个 T，问 T 是不是 S 的子串。

本质不同子串计数。

字典序第 k 小本质不同子串。

后缀自动机

沿着 next 转移能走出的所有字符串即为 S 的子串。

继续回顾自动机小练习。

找到最短的不是 S 子串的串。

后缀自动机

沿着 next 转移能走出的所有字符串即为 S 的子串。

继续回顾自动机小练习。

找到最短的不是 S 子串的串。

最短不公共子串

找到最短的串，使得它是 S 的子串/子序列，但不是 T 的子串/子序列。$n \leq 2000$。

沿着 next 转移能走出的所有字符串即为 S 的子串。

另一个自动机题目：

一道题

给定 S, T。求有多少个字符串能被表示为 S_1T_1，使得 S_1 是 S 的子串，T_1 是 T 的子串。

综合应用

后缀自动机

- 每一个节点对应一些 S 的子串。现在是在某个位置结束，长度在一个区间内的串。
- 树上一个节点的父亲对应它的一个后缀。
- 每个节点维护 $next_c$，表示向后加入字符对应的节点。

我们见过了用单独树边（后缀树）的题目，以及单独用转移边（后缀自动机）的题目，那么自然还有……

你的名字 Easy Version 的一步

给定 S, T。对于 T 的每一个前缀 $T[1:i]$，求出最小的 j 使得 $T[j:i]$ 是 S 的子串。

我们可以像 KMP 匹配前缀一样匹配子串.jpg

Hint: 如果你觉得压缩后的后缀树比较复杂，可以先考虑压缩前的版本。

综合应用

一道题

给定 S，多次询问，每次问 $S[l:r]$ 在 S 的所有本质不同子串中出现次数之和。

目录

写在最后

- 字符串数据结构是一些基本的工具。想要熟练运用，更需要自行多加练习（OJ→ 字符串相关标签 → 刷题)。
- 还有更多更多的工具。一些我听说过名字但没学过的：SA-IS，双向插入后缀树/SAM(Ukkonen's algorithm)，Lyndon 分解，Runs，基本子串字典等等。如果你极其学有余力且感兴趣，那么可以看看。但我相信现在这类科技已经被考纲防出去了。
- 学会用这些工具是必要的，但也不要只想着用工具，要结合具体情况灵活判断。例如，AC 自动机求出现次数的题很多后缀树也能做，但那样是给自己找麻烦。同时，还有一些题就是只靠智慧……

一个小例子

求出 S 中有多少个子串（重复算多次）能被表示成一个串重复两次。

这当然可以后缀树（统计 $LCP(S[i:], S[j:]) \geq |i-j|$，然后可以树剖）。但这样就做不了下面这个题目：

优秀的拆分

给定 S，求出有多少对 $i \leq j < k$，使得 $S[i:j], S[j+1:k]$ 都满足上一题的条件。

Thanks!