

字符串

zjp_shadow

2019 年 1 月 4 日

Preface

Preface

~~竟然被分到这个深坑专题。。。~~

Preface

~~竟然被分到这个深坑专题。。。——~~

具体内容基本就是常见的字符串算法讲解，以及各种题目的套路。

Preface

~~竟然被分到这个深坑专题。。。——~~

具体内容基本就是常见的字符串算法讲解，以及各种题目的套路。

还有一些对于不会造数据的出题人的题目的奇技淫巧。

Contents

- Hash
- KMP & ExKMP(Z-Algorithm)
- Manacher
- Trie
- Aho-Corasick Automaton
- Suffix Array
- Suffix Automaton

Hash

Hash

哈希是一种快速判断字符串（或各种模型），是否相同（匹配）的方式。

Hash

哈希是一种快速判断字符串（或各种模型），是否相同（匹配）的方式。
在字符串中最为主要的应用就是二分哈希求 LCP 。

Hash

哈希是一种快速判断字符串（或各种模型），是否相同（匹配）的方式。

在字符串中最为主要的应用就是二分哈希求 LCP 。

这个在大部分题目，不失为一个十分优秀的暴力（正解）的解法。

Principle

Principle

其主要涉及的思想就是，通过随机化等方法给一个不太容易快速判断相等的模型分配数值或其他相对容易判断相等的哈希值。

Principle

其主要涉及的思想就是，通过随机化等方法给一个不太容易快速判断相等的模型分配数值或其他相对容易判断相等的哈希值。

但是如果哈希值被限制在较小的范围内，并且存在较多的元素，此时由于生日悖论，哈希冲突几率较高。

Principle

其主要涉及的思想就是，通过随机化等方法给一个不太容易快速判断相等的模型分配数值或其他相对容易判断相等的哈希值。

但是如果哈希值被限制在较小的范围内，并且存在较多的元素，此时由于生日悖论，哈希冲突几率较高。

所以一般设计的 Hash 方式都需要满足，哈希值范围远超过元素个数且能较好地体现模型的特殊性。

String

String

常见哈希一个字符串的方式，我们令 $Hash[i]$ 为 S 串到 i 位的哈希值，我们把 p 当做模数。（通常 p 都用 2^{64} 自然溢出）。那么用秦九韶可以得到：

$$Hash[i] = (Hash[i - 1] * Base + S[i]) \mod p$$

String

常见哈希一个字符串的方式，我们令 $Hash[i]$ 为 S 串到 i 位的哈希值，我们把 p 当做模数。（通常 p 都用 2^{64} 自然溢出）。那么用秦九韶可以得到：

$$Hash[i] = (Hash[i-1] * Base + S[i]) \mod p$$

此处 $Base$ 为一个常数，需要比字符集大，通常取 255。

那么从第 i 位向左数第 j 位，它的这位权值就是 $S[i-j] \times Base^j \ j \in [0, i]$ 。

String

常见哈希一个字符串的方式，我们令 $Hash[i]$ 为 S 串到 i 位的哈希值，我们把 p 当做模数。（通常 p 都用 2^{64} 自然溢出）。那么用秦九韶可以得到：

$$Hash[i] = (Hash[i-1] * Base + S[i]) \mod p$$

此处 $Base$ 为一个常数，需要比字符集大，通常取 255。

那么从第 i 位向左数第 j 位，它的这位权值就是 $S[i-j] \times Base^j \ j \in [0, i)$ 。

这样实现有什么好处呢？

String

常见哈希一个字符串的方式，我们令 $Hash[i]$ 为 S 串到 i 位的哈希值，我们把 p 当做模数。（通常 p 都用 2^{64} 自然溢出）。那么用秦九韶可以得到：

$$Hash[i] = (Hash[i-1] * Base + S[i]) \mod p$$

此处 $Base$ 为一个常数，需要比字符集大，通常取 255。

那么从第 i 位向左数第 j 位，它的这位权值就是 $S[i-j] \times Base^j \ j \in [0, i]$ 。

这样实现有什么好处呢？我们可以快速得到一段连续用秦九韶得到的哈希值。

String

常见哈希一个字符串的方式，我们令 $Hash[i]$ 为 S 串到 i 位的哈希值，我们把 p 当做模数。（通常 p 都用 2^{64} 自然溢出）。那么用秦九韶可以得到：

$$Hash[i] = (Hash[i-1] * Base + S[i]) \mod p$$

此处 $Base$ 为一个常数，需要比字符集大，通常取 255。

那么从第 i 位向左数第 j 位，它的这位权值就是 $S[i-j] \times Base^j \ j \in [0, i)$ 。

这样实现有什么好处呢？我们可以快速得到一段连续用秦九韶得到的哈希值。

比如求 $[l, r]$ 这段区间的哈希值，那么就有：

$$(Hash[r] - Hash[l-1] \times Base^{r-l+1}) \mod p$$

String

常见哈希一个字符串的方式，我们令 $Hash[i]$ 为 S 串到 i 位的哈希值，我们把 p 当做模数。（通常 p 都用 2^{64} 自然溢出）。那么用秦九韶可以得到：

$$Hash[i] = (Hash[i-1] * Base + S[i]) \mod p$$

此处 $Base$ 为一个常数，需要比字符集大，通常取 255。

那么从第 i 位向左数第 j 位，它的这位权值就是 $S[i-j] \times Base^j \ j \in [0, i]$ 。

这样实现有什么好处呢？我们可以快速得到一段连续用秦九韶得到的哈希值。

比如求 $[l, r]$ 这段区间的哈希值，那么就有：

$$(Hash[r] - Hash[l-1] \times Base^{r-l+1}) \mod p$$

然后我们就可以得到快速比较两个子串是否相等的方式了。预处理 $O(n)$ ，查询是 $O(1)$ 的。

Tree¹

¹4337: BJOI2015 树的同构

Tree¹

哈希对于许多模型都是十分有效的。比如经典问题比较两个无根树是否同构。

¹4337: BJOI2015 树的同构

Tree¹

哈希对于许多模型都是十分有效的。比如经典问题比较两个无根树是否同构。
如果有根树的话，比较好解决。但是 *Hash* 不能太弱，要体现树的结构。

¹4337: BJOI2015 树的同构

Tree¹

哈希对于许多模型都是十分有效的。比如经典问题比较两个无根树是否同构。

如果是有根树的话，比较好解决。但是 *Hash* 不能太弱，要体现树的结构。

其中一种有效的方式如下：

对于一个节点 u ，先求出它所有儿子 v 的节点的哈希值，然后从小到大排序，记为 H_1, H_2, \dots, H_D 。

那么 u 的哈希值就可以如下计算：

$$\text{Hash}(u) = (\dots(((val \times A) \oplus H_1 \times A) \oplus H_2 \times A) \dots \times A) \oplus H_D + B \pmod{p}$$

¹4337: BJOI2015 树的同构

Tree¹

哈希对于许多模型都是十分有效的。比如经典问题比较两个无根树是否同构。

如果是有根树的话，比较好解决。但是 *Hash* 不能太弱，要体现树的结构。

其中一种有效的方式如下：

对于一个节点 u ，先求出它所有儿子 v 的节点的哈希值，然后从小到大排序，记为 H_1, H_2, \dots, H_D 。

那么 u 的哈希值就可以如下计算：

$$\text{Hash}(u) = (\dots(((val \times A) \oplus H_1 \times A) \oplus H_2 \times A) \dots \times A) \oplus H_D + B \pmod{p}$$

然后我们可以快速判断两个有根树是否同构了。对于无根树，我们可以强制定特殊点为根。例如重心不失为一个较好的解决方案。

但重心可能会有两个，我们取其中哈希值较小的那个的作为这颗树的哈希值即可。

复杂度瓶颈在于排序，所以是 $O(n \log n)$ 的。

¹4337: BJOI2015 树的同构

Graph²

Graph²

有了树哈希，我们自然地想有没有图哈希。

Graph²

有了树哈希，我们自然地想有没有图哈希。

如果只是一次的 *Hash* 显然不是特别靠谱，我们可以考虑对于 *Hash* 进行 *Hash* 。

Graph²

有了树哈希，我们自然地想有没有图哈希。

如果只是一次的 *Hash* 显然不是特别靠谱，我们可以考虑对于 *Hash* 进行 *Hash*。

初始时设每个点为点权为 1，之后进行 n 次迭代。

每次迭代每个点的值更替为与其相邻的点和他本身上一次迭代后的权值计算出的哈希值。

$$\text{Hash}[u][j] = A \times \text{Hash}[u][j-1] + B \times \sum_{(u,v) \in E} (\text{Hash}[v][j-1] + j) \pmod{p}$$

Graph²

有了树哈希，我们自然地想有没有图哈希。

如果只是一次的 *Hash* 显然不是特别靠谱，我们可以考虑对于 *Hash* 进行 *Hash*。

初始时设每个点为点权为 1，之后进行 n 次迭代。

每次迭代每个点的值更替为与其相邻的点和他本身上一次迭代后的权值计算出的哈希值。

$$Hash[u][j] = A \times Hash[u][j-1] + B \times \sum_{(u,v) \in E} (Hash[v][j-1] + j) \pmod{p}$$

然后每次迭代后，我们把所有 *Hash* 值排序，如果对应不同则不同构。

Graph²

有了树哈希，我们自然地想有没有图哈希。

如果只是一次的 *Hash* 显然不是特别靠谱，我们可以考虑对于 *Hash* 进行 *Hash*。

初始时设每个点为点权为 1，之后进行 n 次迭代。

每次迭代每个点的值更替为与其相邻的点和他本身上一次迭代后的权值计算出的哈希值。

$$\text{Hash}[u][j] = A \times \text{Hash}[u][j-1] + B \times \sum_{(u,v) \in E} (\text{Hash}[v][j-1] + j) \pmod{p}$$

然后每次迭代后，我们把所有 *Hash* 值排序，如果对应不同则不同构。

每次的错误率是 $\frac{1}{p}$ ，通过 n 次测试可以认为完全正确。复杂度为 $O(nm + n^2 \log n)$ 。

[CTSC2014] 企鹅 QQ³

给你 N 个长为 L **互不相同** 的字符串，问有多少对字符串满足只有一个位置不同。

$N \leq 30000, L \leq 200$

[CTSC2014] 企鹅 QQ

[CTSC2014] 企鹅 QQ

哈希入门题。

[CTSC2014] 企鹅 QQ

哈希入门题。

可以直接枚举不同的那个位置是什么，然后把 N 个字符串的前缀和后缀的哈希值弄出来。

对于 N 个值进行排序，一遍扫过去的统计两个值都一样的对数。

此处哈希类似前面那样计算即可，时间复杂度是 $O(NL \log N)$ 的。

[CQOI2014] 通配符匹配⁴

给你一个模板串 S ，其中存在两种通配符 $*$, $?$ 。

- $*$ 可以匹配 0 或者任意个字符；
- $?$ 可以且必须匹配一个字符。

有 n 次询问不带通配符的字符串 T ， S 是否能匹配 T 。

$1 \leq n \leq 100, |S|, |T| \leq 10^5$ 通配符数量不超过 10。

⁴LOJ 2242

[CQOI2014] 通配符匹配

[CQOI2014] 通配符匹配

其实可以直接对于正则表达式建出NFA，直接跑。

[CQOI2014] 通配符匹配

其实可以直接对于正则表达式建出NFA，直接跑。

像通配的题，我们通常设 $f_{i,j}$ 表示 S 串到 i 位， T 串到 j 位是否可行，然后可以在 $O(|S||T|)$ 的时间内解决。

[CQOI2014] 通配符匹配

其实可以直接对于正则表达式建出NFA，直接跑。

像通配的题，我们通常设 $f_{i,j}$ 表示 S 串到 i 位， T 串到 j 位是否可行，然后可以在 $O(|S||T|)$ 的时间内解决。

如何优化呢？

[CQOI2014] 通配符匹配

其实可以直接对于正则表达式建出NFA，直接跑。

像通配的题，我们通常设 $f_{i,j}$ 表示 S 串到 i 位， T 串到 j 位是否可行，然后可以在 $O(|S||T|)$ 的时间内解决。

如何优化呢？不难发现对于 S 串来说，其实很多段是可以连续转移的，也就是那些有通配符的地方才有用。

[CQOI2014] 通配符匹配

其实可以直接对于正则表达式建出NFA，直接跑。

像通配的题，我们通常设 $f_{i,j}$ 表示 S 串到 i 位， T 串到 j 位是否可行，然后可以在 $O(|S||T|)$ 的时间内解决。

如何优化呢？不难发现对于 S 串来说，其实很多段是可以连续转移的，也就是那些有通配符的地方才有用。

那么就可以设 $f_{i,j}$ 为 S 串到第 i 个通配符， T 串到 j 位是否可行，对于串尾的 $*$, $?$ 的转移单独考虑。

这样的话，每次转移会转移连续的一段，我们就是要判断对于 S 和 T 对应的两段是否相同，直接用哈希比较即可。复杂度是 $O(|S| + 10 \sum |T|)$ 的。

周期串查询⁵

维护一个长为 n 的只包含数字的字符串，要求支持两个操作，操作共有 m 次。

- 1 $l\ r\ c$ 把 l 到 r 这一段的字符都变成 c 字符；
- 2 $l\ r\ d$ 询问 l 到 r 这一段的子串是否满足以 d 为周期。

字符串 S 是以 x ($1 \leq x \leq |S|$) 为周期的串的条件是：对于所有的 i 从 1 到 $|S| - x$ ， $S_i = S_{i+x}$ 都成立。

$$1 \leq n, m \leq 10^5$$

周期串查询

周期串查询

周期不太好处理，我们可以把它和 *border*（公共前后缀）转化。

周期串查询

周期不太好处理，我们可以把它和 *border*（公共前后缀）转化。

如果字符串 S 有长为 x 的循环节，那么意味着它需要有长为 $|S| - x$ 的 *border*。也就是当 $S_{0 \sim |S| - x} = S_{x \sim |S|}$ 时满足 S 有 x 的循环节。

周期串查询

周期不太好处理，我们可以把它和 *border*（公共前后缀）转化。

如果字符串 S 有长为 x 的循环节，那么意味着它需要有长为 $|S| - x$ 的 *border*。也就是当 $S_{0 \sim |S| - x} = S_{x \sim |S|}$ 时满足 S 有 x 的循环节。

那么问题就转化成区间赋值，区间查询子串是否相等。

周期串查询

周期不太好处理，我们可以把它和 *border*（公共前后缀）转化。

如果字符串 S 有长为 x 的循环节，那么意味着它需要有长为 $|S| - x$ 的 *border*。也就是当 $S_{0 \sim |S| - x} = S_{x \sim |S|}$ 时满足 S 有 x 的循环节。

那么问题就转化成区间赋值，区间查询子串是否相等。

判断子串相等，我们不难想到用哈希实现。那么用线段树维护哈希值即可。区间赋值的话，如果是前面的秦九韶法哈希就赋上 $c \sum_{i=0}^{r-l} \text{Base}^i$ ，然后记得打上标记。

每次查询区间哈希值随意讨论一下情况即可，特判掉 $r - l + 1 = x$ 的情况。

周期串查询

周期不太好处理，我们可以把它和 *border*（公共前后缀）转化。

如果字符串 S 有长为 x 的循环节，那么意味着它需要有长为 $|S| - x$ 的 *border*。也就是当 $S_{0 \sim |S| - x} = S_{x \sim |S|}$ 时满足 S 有 x 的循环节。

那么问题就转化成区间赋值，区间查询子串是否相等。

判断子串相等，我们不难想到用哈希实现。那么用线段树维护哈希值即可。区间赋值的话，如果是前面的秦九韶法哈希就赋上 $c \sum_{i=0}^{r-l} \text{Base}^i$ ，然后记得打上标记。

每次查询区间哈希值随意讨论一下情况即可，特判掉 $r - l + 1 = x$ 的情况。

此处体现了 *Hash* 比别的字符串算法优秀的地方，可以任意修改单点 or 区间字符，查询区间子串是否相同。

KMP

KMP

前面我们知道了有一种字符串匹配算法 *Hash* 。虽然不是特别稳定，但表现不错。

KMP

前面我们知道了有一种字符串匹配算法 *Hash*。虽然不是特别稳定，但表现不错。

那么是否存在一种稳定的字符串匹配算法呢？

KMP

前面我们知道了有一种字符串匹配算法 *Hash* 。虽然不是特别稳定，但表现不错。

那么是否存在一种稳定的字符串匹配算法呢？那就是 KMP 啦。

Principle⁶

⁶一般实现的都是 MP 算法，KMP 需要对失配函数进行优化。

Principle⁶

考虑平常我们是如何暴力匹配两个字符串的，我们用 T 去匹配 S 时，我们逐个匹配直到失配就重新从头匹配。随机数据下表现优秀，期望 $O(|S| + |T|)$ 。但在最坏数据下是 $O(|S||T|)$ 的。

⁶一般实现的都是 MP 算法，KMP 需要对失配函数进行优化。 ◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ 🔍 ↺ ↻

Principle⁶

考虑平常我们是如何暴力匹配两个字符串的，我们用 T 去匹配 S 时，我们逐个匹配直到失配就重新从头匹配。随机数据下表现优秀，期望 $O(|S| + |T|)$ 。但在最坏数据下是 $O(|S||T|)$ 的。

考虑优化，其实每次失配后没有必要从头开始匹配，我们找到一个最靠后且能匹配上的位置就行了。我们定义 $fail_i$ 为 T_i 这一位失配后跳到的位置，那么每次不断向前跳，直到匹配到一个合法的位置即可。

⁶一般实现的都是 MP 算法，KMP 需要对失配函数进行优化。

Principle⁶

考虑平常我们是如何暴力匹配两个字符串的，我们用 T 去匹配 S 时，我们逐个匹配直到失配就重新从头匹配。随机数据下表现优秀，期望 $O(|S| + |T|)$ 。但在最坏数据下是 $O(|S||T|)$ 的。

考虑优化，其实每次失配后没有必要从头开始匹配，我们找到一个最靠后且能匹配上的位置就行了。我们定义 $fail_i$ 为 T_i 这一位失配后跳到的位置，那么每次不断向前跳，直到匹配到一个合法的位置即可。

那么现在只要求出 $fail_i$ 。其实 $fail_i$ 就是以 T_i 为结尾的子串，除去本身最长的 *border* 的长度。

我们可以从 $fail_{i-1}$ 推导到 $fail_i$ ，首先令 $fail_i = fail_{i-1}$ 然后不断判断是否可行，不可行向前跳 *fail*。

⁶一般实现的都是 MP 算法，KMP 需要对失配函数进行优化。

Principle⁶

考虑平常我们是如何暴力匹配两个字符串的，我们用 T 去匹配 S 时，我们逐个匹配直到失配就重新从头匹配。随机数据下表现优秀，期望 $O(|S| + |T|)$ 。但在最坏数据下是 $O(|S||T|)$ 的。

考虑优化，其实每次失配后没有必要从头开始匹配，我们找到一个最靠后且能匹配上的位置就行了。我们定义 $fail_i$ 为 T_i 这一位失配后跳到的位置，那么每次不断向前跳，直到匹配到一个合法的位置即可。

那么现在只要求出 $fail_i$ 。其实 $fail_i$ 就是以 T_i 为结尾的子串，除去本身最长的 *border* 的长度。

我们可以从 $fail_{i-1}$ 推导到 $fail_i$ ，首先令 $fail_i = fail_{i-1}$ 然后不断判断是否可行，不可行向前跳 *fail*。

复杂度是 $O(n)$ 的，已经达到理论的下界。

⁶一般实现的都是 MP 算法，KMP 需要对失配函数进行优化。

Password⁷

给你一个字符串 S ，找到一个最长的串满足它既是 S 的前缀，后缀和非前后缀的一个子串。

比如对于 "fixprefixsuffix"，答案就是 "fix"。无解输出 "Just a legend"。

$$|S| \leq 10^6$$

Password

如果只要满足前后缀，那么 $fail_{|S|}$ 就是我们要求的答案。

Password

如果只要满足前后缀，那么 $fail_{|S|}$ 就是我们要求的答案。

考虑如何修补，如果它是不为前后缀的一个子串并且为前缀，那么意味着它一定对应之前某点 i 的 $fail_i$ 。

那么我们对于每个 i 的 $fail_i$ 标记每个可行的位置。

Password

如果只要满足前后缀，那么 $fail_{|S|}$ 就是我们要求的答案。

考虑如何修补，如果它是不为前后缀的一个子串并且为前缀，那么意味着它一定对应之前某点 i 的 $fail_i$ 。

那么我们对于每个 i 的 $fail_i$ 标记每个可行的位置。

如果 $fail_{|S|}$ 不行，那么下个可行的只能是 $fail_{fail_{|S|}}$ ，我们一直找到第一个被标记的位置就行了，这一定是最长的。

时间复杂度是 $O(n)$ 的。

Best Representation⁸

定义一个**内部不包含循环子串** S 出现次数不少于 2 的串为好串，比如 a 为好串， $abab$ 不为好串， cdc dc 为好串。

给你一个字符串 w ，现在要把它划分成很多段好串，其中段数要尽量少。然后问段数尽量上的前提下，问有多少种划分方式。（对于 $10^9 + 7$ 取模）

$$|w| \leq 5 \times 10^5$$

⁸Atcoder Regular Contest 060 F

Best Representation

Best Representation

判断一个串是否为好串可以先用 *kmp*，求出每个前缀的 *fail(border)*，利用前面讲到的结论知道最小循环串可以和 *border* 转化。

所以它为好串，当且仅当 $fail_i$ 为 0 或者 $i \bmod (i - fail_i) \neq 0$ 。

Best Representation

判断一个串是否为好串可以先用 *kmp*，求出每个前缀的 *fail(border)*，利用前面讲到的结论知道最小循环串可以和 *border* 转化。

所以它为好串，当且仅当 $fail_i$ 为 0 或者 $i \bmod (i - fail_i) \neq 0$ 。

首先可以特殊考虑两种情况（假设最少的段数为 l ）：

- $l = |w|$ ：此时所有字母都是一样的，可以在最后判断。
- $l = 1$ ：此时整个串可行，可以直接判掉。

Best Representation

判断一个串是否为好串可以先用 *kmp*，求出每个前缀的 *fail*(*border*)，利用前面讲到的结论知道最小循环串可以和 *border* 转化。

所以它为好串，当且仅当 $fail_i$ 为 0 或者 $i \bmod (i - fail_i) \neq 0$ 。

首先可以特殊考虑两种情况（假设最少的段数为 l ）：

- $l = |w|$ ：此时所有字母都是一样的，可以在最后判断。
- $l = 1$ ：此时整个串可行，可以直接判掉。

剩下的情况其实答案都为 $l = 2$ ，为什么呢？

Best Representation

判断一个串是否为好串可以先用 *kmp*，求出每个前缀的 *fail(border)*，利用前面讲到的结论知道最小循环串可以和 *border* 转化。

所以它为好串，当且仅当 $fail_i$ 为 0 或者 $i \bmod (i - fail_i) \neq 0$ 。

首先可以特殊考虑两种情况（假设最少的段数为 l ）：

- $l = |w|$ ：此时所有字母都是一样的，可以在最后判断。
- $l = 1$ ：此时整个串可行，可以直接判掉。

剩下的情况其实答案都为 $l = 2$ ，为什么呢？

因为你可以考虑把 $1 \sim |w| - 1$ 分到一组，最后一个字母单独分成一组。

Best Representation

判断一个串是否为好串可以先用 *kmp*，求出每个前缀的 *fail(border)*，利用前面讲到的结论知道最小循环串可以和 *border* 转化。

所以它为好串，当且仅当 $fail_i$ 为 0 或者 $i \bmod (i - fail_i) \neq 0$ 。

首先可以特殊考虑两种情况（假设最少的段数为 l ）：

- $l = |w|$ ：此时所有字母都是一样的，可以在最后判断。
- $l = 1$ ：此时整个串可行，可以直接判掉。

剩下的情况其实答案都为 $l = 2$ ，为什么呢？

因为你可以考虑把 $1 \sim |w| - 1$ 分到一组，最后一个字母单独分成一组。

如果这样不可行的话，只有两种情况：

- 全部字母相等，前面在 $l = |w|$ 时会特判掉。
- $1 \sim |S| - 1$ 存在一个最小循环串 $|T|$ 出现次数不少于 2，那么 $1 \sim |S|$ 一定不存在一个可行的最小循环串，在 $l = 1$ 时会特判掉。

Best Representation

判断一个串是否为好串可以先用 *kmp*，求出每个前缀的 *fail(border)*，利用前面讲到的结论知道最小循环串可以和 *border* 转化。

所以它为好串，当且仅当 $fail_i$ 为 0 或者 $i \bmod (i - fail_i) \neq 0$ 。

首先可以特殊考虑两种情况（假设最少的段数为 l ）：

- $l = |w|$ ：此时所有字母都是一样的，可以在最后判断。
- $l = 1$ ：此时整个串可行，可以直接判掉。

剩下的情况其实答案都为 $l = 2$ ，为什么呢？

因为你可以考虑把 $1 \sim |w| - 1$ 分到一组，最后一个字母单独分成一组。

如果这样不可行的话，只有两种情况：

- 全部字母相等，前面在 $l = |w|$ 时会特判掉。
- $1 \sim |S| - 1$ 存在一个最小循环串 $|T|$ 出现次数不少于 2，那么 $1 \sim |S|$ 一定不存在一个可行的最小循环串，在 $l = 1$ 时会特判掉。

然后剩下计算方案数，只需要枚举断点，然后判断前后缀是否为好串即可。

Best Representation

判断一个串是否为好串可以先用 *kmp*，求出每个前缀的 *fail(border)*，利用前面讲到的结论知道最小循环串可以和 *border* 转化。

所以它为好串，当且仅当 $fail_i$ 为 0 或者 $i \bmod (i - fail_i) \neq 0$ 。

首先可以特殊考虑两种情况（假设最少的段数为 l ）：

- $l = |w|$ ：此时所有字母都是一样的，可以在最后判断。
- $l = 1$ ：此时整个串可行，可以直接判掉。

剩下的情况其实答案都为 $l = 2$ ，为什么呢？

因为你可以考虑把 $1 \sim |w| - 1$ 分到一组，最后一个字母单独分成一组。

如果这样不可行的话，只有两种情况：

- 全部字母相等，前面在 $l = |w|$ 时会特判掉。
- $1 \sim |S| - 1$ 存在一个最小循环串 $|T|$ 出现次数不少于 2，那么 $1 \sim |S|$ 一定不存在一个可行的最小循环串，在 $l = 1$ 时会特判掉。

然后剩下计算方案数，只需要枚举断点，然后判断前后缀是否为好串即可。

所以模数是唬你的 2333

[ExKMP] 最长共同前缀长度⁹

⁹CaiOJ 1461

[ExKMP] 最长共同前缀长度⁹

众所周知，KMP 算法是最为经典的单模板字符串匹配问题的线性解法。
那么 ExKMP 字面意义是 KMP 的扩展，那么它是解决什么问题呢？

⁹CaiOJ 1461

[ExKMP] 最长共同前缀长度⁹

众所周知，KMP 算法是最为经典的单模板字符串匹配问题的线性解法。
那么 ExKMP 字面意义是 KMP 的扩展，那么它是解决什么问题呢？

存在母串 S 和子串 T ，设 $|S| = n, |T| = m$ ，求 T 与 S 的每一个后缀的最长公共前缀 (LCP)。设 $extend$ 数组， $extend[i]$ 表示 T 与 $S_{i \sim n}$ 的 LCP，对于 $i \in [1, n]$ 求 $extend[i]$ 。

$1 \leq m \leq n \leq 10^6$

Get extend

¹⁰<https://www.cnblogs.com/zjp-shadow/p/10139818.html>

Get extend

其实可以直接用 SA/SAM 解决，用 $height$ 直接扫一遍就行了，但是太大材小用了。。。

¹⁰<https://www.cnblogs.com/zjp-shadow/p/10139818.html>

Get extend

其实可以直接用 *SA/SAM* 解决，用 *height* 直接扫一遍就行了，但是太大材小用了。。。

我们需要一个辅助的匹配数组 $next[i]$ 表示 $T_{i \sim m}$ 与 T 的 LCP。

此处假设我们已经得到了 $next[i]$ ，当前我们从前往后依次递推 $extend[i]$ ，假设当前递推完前 k 位，要求 $k+1$ 位。

此时 $extend[1 \sim k]$ 已经算完，假设之前 T 能匹配 S 的后缀最远的位置为

$p = \max_{i < k} (i + extend[i] - 1)$ ，对应取到最大值的位置 i 为 pos 。

Get extend

其实可以直接用 *SA/SAM* 解决，用 *height* 直接扫一遍就行了，但是太大材小用了。。。

我们需要一个辅助的匹配数组 $next[i]$ 表示 $T_{i \sim m}$ 与 T 的 LCP。

此处假设我们已经得到了 $next[i]$ ，当前我们从前往后依次递推 $extend[i]$ ，假设当前递推完前 k 位，要求 $k+1$ 位。

此时 $extend[1 \sim k]$ 已经算完，假设之前 T 能匹配 S 的后缀最远的位置为

$p = \max_{i < k} (i + extend[i] - 1)$ ，对应取到最大值的位置 i 为 pos 。

令 $len = next[k - pos + 2]$ ，分以下两种情况讨论。

Get extend

其实可以直接用 *SA/SAM* 解决，用 *height* 直接扫一遍就行了，但是太大材小用了。。。

我们需要一个辅助的匹配数组 $next[i]$ 表示 $T_{i \sim m}$ 与 T 的 LCP。

此处假设我们已经得到了 $next[i]$ ，当前我们从前往后依次递推 $extend[i]$ ，假设当前递推完前 k 位，要求 $k+1$ 位。

此时 $extend[1 \sim k]$ 已经算完，假设之前 T 能匹配 S 的后缀最远的位置为

$p = \max_{i < k} (i + extend[i] - 1)$ ，对应取到最大值的位置 i 为 pos 。

令 $len = next[k - pos + 2]$ ，分以下两种情况讨论。

- $k + len < p$ 此时 $extend[k + 1] = len$ 。
- $k + len \geq p$ 那么 S_{p+1} 之后的串我们都从未尝试匹配过，不知道其信息，我们直接暴力向后依次匹配即可，直到失配停下来。如果 $extend[k + 1] + k > p$ 要更新 p 和 pos 。

Get extend

其实可以直接用 SA/SAM 解决，用 *height* 直接扫一遍就行了，但是太大材小用了。。。

我们需要一个辅助的匹配数组 $next[i]$ 表示 $T_{i \sim m}$ 与 T 的 LCP。

此处假设我们已经得到了 $next[i]$ ，当前我们从前往后依次递推 $extend[i]$ ，假设当前递推完前 k 位，要求 $k+1$ 位。

此时 $extend[1 \sim k]$ 已经算完，假设之前 T 能匹配 S 的后缀最远的位置为

$p = \max_{i < k} (i + extend[i] - 1)$ ，对应取到最大值的位置 i 为 pos 。

令 $len = next[k - pos + 2]$ ，分以下两种情况讨论。

- $k + len < p$ 此时 $extend[k + 1] = len$ 。
- $k + len \geq p$ 那么 S_{p+1} 之后的串我们都从未尝试匹配过，不知道其信息，我们直接暴力向后依次匹配即可，直到失配停下来。如果 $extend[k + 1] + k > p$ 要更新 p 和 pos 。

然后这样就可以匹配完了，具体细节与证明可以参考博客¹⁰。

¹⁰<https://www.cnblogs.com/zjp-shadow/p/10139818.html>

Get next

Get next

前面我们假设已经求出 $next$ ，但如何求呢？

Get next

前面我们假设已经求出 $next$ ，但如何求呢？

其实和 KMP 是很类似的，我们相当于 T 自己匹配自己每个后缀的答案，此处需要的 $next$ 全都在前面会计算过。

所以这里和前面匹配的过程是一模一样的。

复杂度证明

复杂度证明

下面来分析一下算法的时间复杂度。

复杂度证明

下面来分析一下算法的时间复杂度。

对于第一种情况，无需做任何匹配即可计算出 $extend[i]$ 。

复杂度证明

下面来分析一下算法的时间复杂度。

对于第一种情况，无需做任何匹配即可计算出 $extend[i]$ 。

对于第二种情况，都是从未被匹配的位置开始匹配，匹配过的位置不再匹配，也就是说对于母串的每一个位置，都只匹配了一次，所以算法总体时间复杂度是 $O(n)$ 的。

[NOI2014] 动物园¹¹

给你一个字符串 S 。对于字符串 S 的前 i 个字符构成的子串，既是它的后缀同时又是它的前缀，并且 **该后缀与该前缀不重叠**，将这种子串的数量记作 $num[i]$ 。

求

$$\prod_{i=1}^{|S|} (num[i] + 1) \pmod{10^9 + 7}$$

$$|S| \leq 10^6$$

[NOI2014] 动物园

[NOI2014] 动物园

如果会 ExKMP 就是裸题了。

[NOI2014] 动物园

如果会 ExKMP 就是裸题了。

然后考虑对于每个 S 的后缀 i 会被算多少遍，其实就是对于以 $[i, \min(2 \times (i - 1), i + \text{next}[i] - 1)]$ 为结尾的所有前缀有贡献，那么直接差分即可。

复杂度是 $O(\sum |S|)$ 的。

Vasya and Big Integers¹²

给你一个由数字构成的字符串 a ，问你有多少种划分方式，使得每段**不含前导 0**，并且每段的数字大小在 $[l, r]$ 之间。答案对于 998244353 取模。

$$1 \leq a \leq 10^{10^6}, 0 \leq l \leq r \leq 10^{10^6}$$

Vasya and Big Integers

Vasya and Big Integers

考虑暴力 dp ，令 dp_i 为以 i 为一段结束的方案数。对于填表法是没有那么好转移的，（因为前导 0 的限制是挂在前面那个点上）我们考虑**刷表法**。

那么转移为

$$dp_j = dp_j + dp_i \quad \{j \mid a_i \neq 0 \& l \leq a_{i \sim j} \leq r\}$$

Vasya and Big Integers

考虑暴力 dp ，令 dp_i 为以 i 为一段结束的方案数。对于填表法是没有那么好转移的，（因为前导 0 的限制是挂在前面那个点上）我们考虑**刷表法**。

那么转移为

$$dp_j = dp_j + dp_i \quad \{j \mid a_i \neq 0 \& l \leq a_{i \sim j} \leq r\}$$

我们发现 dp_i 能转移到的 j 一定是一段**连续的区间**。

我们就需要快速得到这段区间，首先不难发现 j 对应的位数区间是可以很快确定的，就是 $[i + |L| - 1, i + |R| - 1]$ 。

Vasya and Big Integers

考虑暴力 dp ，令 dp_i 为以 i 为一段结束的方案数。对于填表法是没有那么好转移的，（因为前导 0 的限制是挂在前面那个点上）我们考虑**刷表法**。

那么转移为

$$dp_j = dp_j + dp_i \quad \{j \mid a_i \neq 0 \& l \leq a_{i \sim j} \leq r\}$$

我们发现 dp_i 能转移到的 j 一定是一段**连续的区间**。

我们就需要快速得到这段区间，首先不难发现 j 对应的位数区间是可以很快确定的，就是 $[i + |L| - 1, i + |R| - 1]$ 。

但是如果位数一样的话需要多花费 $O(n)$ 的时间去逐位比较大小。有什么快速的方法吗？

Vasya and Big Integers

考虑暴力 dp ，令 dp_i 为以 i 为一段结束的方案数。对于填表法是没有那么好转移的，（因为前导 0 的限制是挂在前面那个点上）我们考虑**刷表法**。

那么转移为

$$dp_j = dp_j + dp_i \quad \{j \mid a_i \neq 0 \& l \leq a_{i \sim j} \leq r\}$$

我们发现 dp_i 能转移到的 j 一定是一段**连续的区间**。

我们就需要快速得到这段区间，首先不难发现 j 对应的位数区间是可以很快确定的，就是 $[i + |L| - 1, i + |R| - 1]$ 。

但是如果位数一样的话需要多花费 $O(n)$ 的时间去逐位比较大小。有什么快速的方法吗？

不难想到比较两个数字大小的时候是和字符串一样的，就是 LCP 的后面一位。那么我们用 ExKMP 快速预处理 $extend(LCP)$ 就可以了。

Manacher

Manacher

~~没有太多时间准备了。。。大家可以自己去网上找博客和题做。。。~~

Manacher

~~没有太多时间准备了。。。大家可以自己去网上找博客和题做。。。——~~

它是一个可以在 $O(n)$ 的时间内求出对于一个字符串以任意点为对称中心的回文子串的算法。

Manacher

~~没有太多时间准备了。。。大家可以自己去网上找博客和题做。。。——~~

它是一个可以在 $O(n)$ 的时间内求出对于一个字符串以任意点为对称中心的回文子串的算法。

至于题目？大部分的题目都不差二分哈希的那个 \log 。。

DFA

DFA

介绍 *Trie* 等系列自动机之前，一定要介绍一下它们的原理，也就是**有限自动机 (FA)**。

DFA

介绍 *Trie* 等系列自动机之前，一定要介绍一下它们的原理，也就是**有限自动机 (FA)**。

我们通常也只用确定有限状态自动机 (DFA) 解决大部分字符串的问题。

DFA

介绍 *Trie* 等系列自动机之前，一定要介绍一下它们的原理，也就是**有限自动机 (FA)**。

我们通常也只用确定有限状态自动机 (DFA) 解决大部分字符串的问题。

DFA 可以用一个 5 元组 $(Q, \Sigma, \delta, q_0, F)$ 表示，其中 Q 为状态集， Σ 为字母表， δ 为转移函数， q_0 为起始状态， F 为终态集。

DFA

介绍 *Trie* 等系列自动机之前，一定要介绍一下它们的原理，也就是**有限自动机 (FA)**。

我们通常也只用确定有限状态自动机 (DFA) 解决大部分字符串的问题。

DFA 可以用一个 5 元组 $(Q, \Sigma, \delta, q_0, F)$ 表示，其中 Q 为状态集, Σ 为字母表, δ 为转移函数, q_0 为起始状态, F 为终态集。

如何判断一个字符串是否能被一个 DFA 接受呢？

DFA

介绍 *Trie* 等系列自动机之前，一定要介绍一下它们的原理，也就是**有限自动机 (FA)**。

我们通常也只用确定有限状态自动机 (DFA) 解决大部分字符串的问题。

DFA 可以用一个 5 元组 $(Q, \Sigma, \delta, q_0, F)$ 表示，其中 Q 为状态集, Σ 为字母表, δ 为转移函数, q_0 为起始状态, F 为终态集。

如何判断一个字符串是否能被一个 DFA 接受呢？一开始时，自动机在起始状态 q_0 ，每读入一个字符 c 后，状态转移到 $\delta(q, c)$ ，其中 q 为当前状态。当整个字符串读完之后，当且仅当 q 在终态集 F 中时，DFA 接受这个字符串。

Trie

Trie

Trie 图是一种最简易的 DFA 。

Trie

Trie 图是一种最简易的 DFA 。

插入每个模式串的时候，从前往后依次考虑每一位 c ，然后记当前所在的状态 q ，如果不存在 $\delta(q, c)$ 那么添加一个 $\delta(q, c)$ 的转移。接下来 $q \rightarrow \delta(q, c)$ 。终态的 q 作为 DFA 的一个 F 的一个元素。

Trie

Trie 图是一种最简易的 DFA 。

插入每个模式串的时候，从前往后依次考虑每一位 c ，然后记当前所在的状态 q ，如果不存在 $\delta(q, c)$ 那么添加一个 $\delta(q, c)$ 的转移。接下来 $q \rightarrow \delta(q, c)$ 。终态的 q 作为 DFA 的一个 F 的一个元素。

不难发现 *Trie* 能接受的所有字符串就是插入的所有的串。

Trie

Trie 图是一种最简易的 DFA 。

插入每个模式串的时候，从前往后依次考虑每一位 c ，然后记当前所在的状态 q ，如果不存在 $\delta(q, c)$ 那么添加一个 $\delta(q, c)$ 的转移。接下来 $q \rightarrow \delta(q, c)$ 。终态的 q 作为 DFA 的一个 F 的一个元素。

不难发现 *Trie* 能接受的所有字符串就是插入的所有的串。

因为这次是主要讲字符串就不提 01trie 在异或和数据结构中的运用了。

Polycarp's phone book¹³

有 n 个长度为 9 且只包含数字字符且**互不相同**的串。

需要对于每个串找到一个**长度最短**的识别码，使得这个识别码**当且仅当**为这个串的子串。

$$n \leq 70000$$

Polycarp's phone book

一道 *Trie* 入门题。。

Polycarp's phone book

一道 *Trie* 入门题。。

因为长度不长，考虑把每个串的后缀都插入 *Trie* 中，然后统计每个串的出现次数。

全都插入后，最后对于每个串枚举它的后缀然后在 *Trie* 上找到第一个出现次数为 1 的子串，然后取一个长度最短的串作为答案即可。

Polycarp's phone book

一道 *Trie* 入门题。。

因为长度不长，考虑把每个串的后缀都插入 *Trie* 中，然后统计每个串的出现次数。

全都插入后，最后对于每个串枚举它的后缀然后在 *Trie* 上找到第一个出现次数为 1 的子串，然后取一个长度最短的串作为答案即可。

但是这样会有点小问题。比如对于 100000000 来说，0 会被统计 8 次，显然不合理。那么对于同样一个串的子串我们记一个时间戳，这样就可以保证只统计一个串的本质不同的串。

复杂度是 $O(36n)$ 的。

A Lot of Games¹⁴

给你 n 个字符串，有这样一个游戏。

初始有一个空串 S ，两个人轮流在串后面添加一个字符 c ，任意时刻都要满足 S 是 n 个字符串其中的一个前缀。

不能操作的那个人算作输了这局游戏。

这种游戏一共进行 k 轮，这一场输的下一场可以先手操作。最后的胜负只看**最后一局的胜负**。问先后手谁能必胜。

¹⁴Codeforces 455B

A Lot of Games

其实操作的本质就是把 n 个字符串构成 *Trie*，然后将节点 u 从 *root* 开始向 $\delta(u, c)$ 轮流移动一步，当 $u \in F$ 时需要操作的那个人算输。

A Lot of Games

其实操作的本质就是把 n 个字符串构成 *Trie*，然后将节点 u 从 *root* 开始向 $\delta(u, c)$ 轮流移动一步，当 $u \in F$ 时需要操作的那个人算输。

在 *DAG* 上博弈通常都用 *SG* 函数。但是这样只能判断是否先手必胜。我们有时候需要以退为进，让掉这局以获得下局的先手。

A Lot of Games

其实操作的本质就是把 n 个字符串构成 *Trie*，然后将节点 u 从 *root* 开始向 $\delta(u, c)$ 轮流移动一步，当 $u \in F$ 时需要操作的那个人算输。

在 *DAG* 上博弈通常都用 *SG* 函数。但是这样只能判断是否先手必胜。我们有时候需要以退为进，让掉这局以获得下局的先手。

我们设 $win[u], lose[u]$ 为 u 这个点是否有必胜、必败策略。转移为 $win[u] = !win[v]$ ， $lose$ 同理。

A Lot of Games

其实操作的本质就是把 n 个字符串构成 *Trie*，然后将节点 u 从 *root* 开始向 $\delta(u, c)$ 轮流移动一步，当 $u \in F$ 时需要操作的那个人算输。

在 *DAG* 上博弈通常都用 *SG* 函数。但是这样只能判断是否先手必胜。我们有时候需要以退为进，让掉这局以获得下局的先手。

我们设 $win[u], lose[u]$ 为 u 这个点是否有必胜、必败策略。转移为 $win[u] = !win[v]$ ， $lose$ 同理。

最后我们分三种情况讨论：

- $win[root] = false$ 后手必胜：后手一直让它输，那么先手永远是先手。
- $win[root] = true \& lose[root] = true$ 先手必胜：前 $k-1$ 局都让自己输，最后一局赢即可。
- $win[root] = true \& lose[root] = false$ 当 $k \bmod 2 = 1$ 时先手必胜，否则后手必胜。因为先手不能输，那么每场过后先后手互换，所以最后判断 k 的奇偶就行了。

A Lot of Games

其实操作的本质就是把 n 个字符串构成 *Trie*，然后将节点 u 从 *root* 开始向 $\delta(u, c)$ 轮流移动一步，当 $u \in F$ 时需要操作的那个人算输。

在 *DAG* 上博弈通常都用 *SG* 函数。但是这样只能判断是否先手必胜。我们有时候需要以退为进，让掉这局以获得下局的先手。

我们设 $win[u], lose[u]$ 为 u 这个点是否有必胜、必败策略。转移为 $win[u] = !win[v]$ ， $lose$ 同理。

最后我们分三种情况讨论：

- $win[root] = false$ 后手必胜：后手一直让它输，那么先手永远是先手。
- $win[root] = true \& lose[root] = true$ 先手必胜：前 $k-1$ 局都让自己输，最后一局赢即可。
- $win[root] = true \& lose[root] = false$ 当 $k \bmod 2 = 1$ 时先手必胜，否则后手必胜。因为先手不能输，那么每场过后先后手互换，所以最后判断 k 的奇偶就行了。

~~这其实是道博弈题。。。——~~

Aho-Corasick Automaton

Aho-Corasick Automaton

前面我们讲解了 *Trie*&*KMP*，这两个算法是 *AC* 自动机所需要的前置知识。

Aho-Corasick Automaton

前面我们讲解了 *Trie*&*KMP*，这两个算法是 *AC* 自动机所需要的前置知识。
KMP 是解决单模板匹配的有效算法，那么对于多模板匹配就需要前面介绍的 *DFA* 了。

Aho-Corasick Automaton

前面我们讲解了 *Trie*&*KMP*，这两个算法是 *AC* 自动机所需要的前置知识。
KMP 是解决单模板匹配的有效算法，那么对于多模板匹配就需要前面介绍的 *DFA* 了。

但在 *OI* 竞赛中匹配不是它的本职，计数才是它运用最广的地方 2333

构造方法

构造方法

首先把所有串都插入到 *Trie* 中。

构造方法

首先把所有串都插入到 *Trie* 中。

然后考虑构造它的失配函数 *fail* , $fail_u$ 定义为如果 u 失配后回到的最深深度的节点。也可以理解成 $fail_u$ 对应节点的字符串是 u 的字符串的最长后缀。

构造方法

首先把所有串都插入到 $Trie$ 中。

然后考虑构造它的失配函数 $fail$, $fail_u$ 定义为如果 u 失配后回到的最深深度的节点。也可以理解成 $fail_u$ 对应节点的字符串是 u 的字符串的最长后缀。

因为要层层推进, 所以用 Bfs 遍历整颗 $Trie$ 。

假设当前遍历到状态 u , $\forall c, \delta(u, c)$ 如果存在的话。那么 $fail_{\delta(u, c)} = \delta(fail_u, c)$

。

构造方法

首先把所有串都插入到 *Trie* 中。

然后考虑构造它的失配函数 *fail* , $fail_u$ 定义为如果 u 失配后回到的最深深度的节点。也可以理解成 $fail_u$ 对应节点的字符串是 u 的字符串的最长后缀。

因为要层层推进, 所以用 *Bfs* 遍历整颗 *Trie* 。

假设当前遍历到状态 u , $\forall c, \delta(u, c)$ 如果存在的话。那么 $fail_{\delta(u, c)} = \delta(fail_u, c)$ 。

如果不存在直接令 $\delta(u, c) = \delta(fail_u, c)$ 也就是直接失配后就直接匹配下个地方。也就是把失配的那条边直接建出来。

构造方法

首先把所有串都插入到 *Trie* 中。

然后考虑构造它的失配函数 *fail* , $fail_u$ 定义为如果 u 失配后回到的最深深度的节点。也可以理解成 $fail_u$ 对应节点的字符串是 u 的字符串的最长后缀。

因为要层层推进, 所以用 *Bfs* 遍历整颗 *Trie* 。

假设当前遍历到状态 u , $\forall c, \delta(u, c)$ 如果存在的话。那么 $fail_{\delta(u, c)} = \delta(fail_u, c)$ 。

如果不存在直接令 $\delta(u, c) = \delta(fail_u, c)$ 也就是直接失配后就直接匹配下个地方。也就是把失配的那条边直接建出来。

最后的复杂度就是 $O((\sum |S|) \times |\Sigma|)$ 。

[JSOI2009] 密码¹⁵

有 n 个长度不超过 10 只有小写字母的字符串。

问有多少个长为 L 的字符串包含了前面 n 个字符串。

如果答案 ≤ 42 输出方案。

$$1 \leq L \leq 25, 1 \leq n \leq 10$$

¹⁵BZOJ 1559; WorldFinal 2008, LA 4126

[JSOI2009] 密码

[JSOI2009] 密码

一般这种关于是否包含字符串的计数题，一般都设二维状态。令 $dp[i][j]$ 为字符串到 i 位，在 AC 自动机上第 j 个节点的方案数。

[JSOI2009] 密码

一般这种关于是否包含字符串的计数题，一般都设二维状态。令 $dp[i][j]$ 为字符串到 i 位，在 AC 自动机上第 j 个节点的方案数。

由于它要包含所有字符串，我们多设个状态来状压已经匹配了哪些节点。令 $dp[i][j][k]$ 为字符串到 i 位，在 AC 自动机上第 j 个节点，匹配了的集合为 k 的方案数。

然后我们需要预处理 AC 自动机每个节点能匹配的状态是什么。我们一开始插入的时候打在最后一个节点，最后我们把 u 的状态集合或上 $fail_u$ 的集合就行了。

每次转移的时候沿着 $\delta(j, c)$ 转移即可。复杂度是 $O(nL2^n \times |\Sigma|)$ 的。

[JSOI2009] 密码

一般这种关于是否包含字符串的计数题，一般都设二维状态。令 $dp[i][j]$ 为字符串到 i 位，在 AC 自动机上第 j 个节点的方案数。

由于它要包含所有字符串，我们多设个状态来状压已经匹配了哪些节点。令 $dp[i][j][k]$ 为字符串到 i 位，在 AC 自动机上第 j 个节点，匹配了的集合为 k 的方案数。

然后我们需要预处理 AC 自动机每个节点能匹配的状态是什么。我们一开始插入的时候打在最后一个节点，最后我们把 u 的状态集合或上 $fail_u$ 的集合就行了。

每次转移的时候沿着 $\delta(j, c)$ 转移即可。复杂度是 $O(nL2^n \times |\Sigma|)$ 的。

最后只需要考虑 ≤ 42 如何输出方案即可。肯定不能 $O(|\Sigma|^L)$ 暴力枚举，考虑优化。由于答案 ≤ 42 所以不可能存在两个单词之间存在不需要的字符（不然至少为 $|\Sigma| \times 2$ ）。直接从前往后枚举就行了，然后处理下前后缀的交接就行了。

[JSOI2009] 密码

一般这种关于是否包含字符串的计数题，一般都设二维状态。令 $dp[i][j]$ 为字符串到 i 位，在 AC 自动机上第 j 个节点的方案数。

由于它要包含所有字符串，我们多设个状态来状压已经匹配了哪些节点。令 $dp[i][j][k]$ 为字符串到 i 位，在 AC 自动机上第 j 个节点，匹配了的集合为 k 的方案数。

然后我们需要预处理 AC 自动机每个节点能匹配的状态是什么。我们一开始插入的时候打在最后一个节点，最后我们把 u 的状态集合或上 $fail_u$ 的集合就行了。

每次转移的时候沿着 $\delta(j, c)$ 转移即可。复杂度是 $O(nL2^n \times |\Sigma|)$ 的。

最后只需要考虑 ≤ 42 如何输出方案即可。肯定不能 $O(|\Sigma|^L)$ 暴力枚举，考虑优化。由于答案 ≤ 42 所以不可能存在两个单词之间存在不需要的字符（不然至少为 $|\Sigma| \times 2$ ）。直接从前往后枚举就行了，然后处理下前后缀的交接就行了。

其实还有一种奇技淫巧就是直接在 dp 那里开个 `vector` 记下当前的方案。如果 $dp > 42$ 那么清空 `vector` 并且之后不会进行转移。卡卡空间能在 Luogu 的 128MB 通过，BZOJ 的 64MB 好像卡不过。

e-Government¹⁶

给你 n 个字符串 S ，作为初始的字符串集合。

需要支持共 m 个操作，分为如下三种：

- $+id$ ：把第 id 个字符串加入到集合中，如果存在就忽略。
- $-id$ ：把第 id 个字符串从集合中删除，如果不存在就忽略。
- $?T$ ：查询集合中每个字符串 S 在字符串 T 中出现的次数总和。

$$n, m \leq 10^5, \sum |S|, \sum |T| \leq 10^6$$

e-Government

¹⁷ 请注意，对于非特意构造的随机数据表现十分优秀

e-Government

统计一些串在另外一个串 T 中出现的次数是 AC 自动机的常用功能。

¹⁷ 请注意，对于非特意构造的随机数据表现十分优秀

e-Government

统计一些串在另外一个串 T 中出现的次数是 AC 自动机的常用功能。

具体来说把前面的一些串一起插到 AC 自动机，然后你考虑在 T 在 AC 自动机上沿着 δ 遍历。

假设我们到点 u ，当前的状态可以算作 T 的一个前缀在 AC 自动机上等价的节点。然后你就可以考虑查询所有后缀的贡献。

不难发现就是对于点 u 不断跳 $fail$ 到达的所有点 v 对于 u 的贡献。（ v 对应的字符串就是 u 在 AC 自动机上的最长后缀）

¹⁷ 请注意，对于非特意构造的随机数据表现十分优秀

e-Government

统计一些串在另外一个串 T 中出现的次数是 AC 自动机的常用功能。

具体来说把前面的一些串一起插到 AC 自动机，然后你考虑在 T 在 AC 自动机上沿着 δ 遍历。

假设我们到点 u ，当前的状态可以算作 T 的一个前缀在 AC 自动机上等价的节点。然后你就可以考虑查询所有后缀的贡献。

不难发现就是对于点 u 不断跳 $fail$ 到达的所有点 v 对于 u 的贡献。（ v 对应的字符串就是 u 在 AC 自动机上的最长后缀）

我们如果暴力做过程每次可能是 $O(\sum |S|)$ 的¹⁷。

¹⁷ 请注意，对于非特意构造的随机数据表现十分优秀

e-Government

统计一些串在另外一个串 T 中出现的次数是 AC 自动机的常用功能。

具体来说把前面的一些串一起插到 AC 自动机，然后你考虑在 T 在 AC 自动机上沿着 δ 遍历。

假设我们到点 u ，当前的状态可以算作 T 的一个前缀在 AC 自动机上等价的节点。然后你就可以考虑查询所有后缀的贡献。

不难发现就是对于点 u 不断跳 $fail$ 到达的所有点 v 对于 u 的贡献。（ v 对应的字符串就是 u 在 AC 自动机上的最长后缀）

我们如果暴力做过程每次可能是 $O(\sum |S|)$ 的¹⁷。

考虑这个操作的本质，其实就是对于 $fail$ 反着建出的树上，它到根节点的贡献之和。那么这个题就转化成了树上问题。

需要支持两个操作，单点修改，链上查询。这样我们就可以转化成子树修改，单点查询。那么利用树状数组配合 dfs 序解决即可。

最后复杂度就是 $O(\sum |S| \times |\Sigma| + \sum |T| \log(\sum |S|))$ 。

¹⁷请注意，对于非特意构造的随机数据表现十分优秀

SuffixArray

SuffixArray

后缀数组常用于解决 LCP(LCS) 和一些有关字符串匹配的问题。

SuffixArray

后缀数组常用于解决 LCP(LCS) 和一些有关字符串匹配的问题。
这个数据结构思维难度不是很大，但对于实现能力有较高要求。

SuffixArray

后缀数组常用于解决 LCP(LCS) 和一些有关字符串匹配的问题。
这个数据结构思维难度不是很大，但对于实现能力有较高要求。
简单介绍一下各种常见的构造方式，以及一些相关题目。

Principle

Principle

首先弄清楚后缀数组是个啥子东西。

Principle

首先弄清楚后缀数组是个啥子东西。

我们把一个字符串 S 的所有后缀按字典序大小从小到大排序，然后得到的编号序列就是后缀数组 sa 。

Principle

首先弄清楚后缀数组是个啥子东西。

我们把一个字符串 S 的所有后缀按字典序大小从小到大排序，然后得到的编号序列就是后缀数组 sa 。

转化之后可以保存子串之间有关于 LCP 的信息，解决各种各样的统计子串数的问题

Principle

首先弄清楚后缀数组是个啥子东西。

我们把一个字符串 S 的所有后缀按字典序大小从小到大排序，然后得到的编号序列就是后缀数组 sa 。

转化之后可以保存子串之间有关于 LCP 的信息，解决各种各样的统计子串数的问题

sa 暴力构造是 $O(n^2 \log n)$ 的，不够优秀。下面介绍几种复杂度比较优秀的构造方式。

二分哈希

二分哈希

介绍一种很思博，但考试可以救命的算法。

二分哈希

介绍一种很思博，但考试可以救命的算法。

考虑之前的暴力做法劣在何处。其实就是比较大小需要 $O(n = |S|)$ 的时间。

二分哈希

介绍一种很思博，但考试可以救命的算法。

考虑之前的暴力做法劣在何处。其实就是比较大小需要 $O(n = |S|)$ 的时间。

如何优化呢？

二分哈希

介绍一种很思博，但考试可以救命的算法。

考虑之前的暴力做法劣在何处。其实就是比较大小需要 $O(n = |S|)$ 的时间。

如何优化呢？前面提到了比较字符串大小可以利用二分哈希求 LCP 然后比较。

二分哈希

介绍一种很思博，但考试可以救命的算法。

考虑之前的暴力做法劣在何处。其实就是比较大小需要 $O(n = |S|)$ 的时间。

如何优化呢？前面提到了比较字符串大小可以利用二分哈希求 LCP 然后比较。

然后比较 LCP 的后面第一位的大小就行了。

二分哈希

介绍一种很思博，但考试可以救命的算法。

考虑之前的暴力做法劣在何处。其实就是比较大小需要 $O(n = |S|)$ 的时间。

如何优化呢？前面提到了比较字符串大小可以利用二分哈希求 LCP 然后比较。

然后比较 LCP 的后面第一位的大小就行了。

然后利用 sort，复杂度是 $O(n \log^2 n)$ 的，但可能会被卡哈希。

倍增

接下来介绍一下倍增算法。仍然是优化字符串比较的复杂度。

倍增

接下来介绍一下倍增算法。仍然是优化字符串比较的复杂度。

预处理出前缀和后缀的大小关系，用这两段拼起来比较它们的大小。

倍增

接下来介绍一下倍增算法。仍然是优化字符串比较的复杂度。

预处理出前缀和后缀的大小关系，用这两段拼起来比较它们的大小。

倍增的思路大概就是如此，每次比较长为 2^i 的长度的所有后缀的大小的时候，利用长为 2^{i-1} 的信息来进行优化。直到所有串的名次互不相同即可，这个过程最多进行 $\lceil \log_2 n \rceil$ 次。

倍增

接下来介绍一下倍增算法。仍然是优化字符串比较的复杂度。

预处理出前缀和后缀的大小关系，用这两段拼起来比较它们的大小。

倍增的思路大概就是如此，每次比较长为 2^i 的长度的所有后缀的大小的时候，利用长为 2^{i-1} 的信息来进行优化。直到所有串的名次互不相同即可，这个过程最多进行 $\lceil \log_2 n \rceil$ 次。

然后这个问题就转化成，一个双关键字排序的经典问题，第一关键字就是前半部分的排名，第二关键字就是后半部分的排名。

倍增

接下来介绍一下倍增算法。仍然是优化字符串比较的复杂度。

预处理出前缀和后缀的大小关系，用这两段拼起来比较它们的大小。

倍增的思路大概就是如此，每次比较长为 2^i 的长度的所有后缀的大小的时候，利用长为 2^{i-1} 的信息来进行优化。直到所有串的名次互不相同即可，这个过程最多进行 $\lceil \log_2 n \rceil$ 次。

然后这个问题就转化成，一个双关键字排序的经典问题，第一关键字就是前半部分的排名，第二关键字就是后半部分的排名。

这个我们可以用基数排序把排序复杂度优化到 $O(n)$ ，所以最后的复杂度就是 $O(n \log n)$ 的。

DC3

DC3

简单普及一下一种线性构造算法 DC3。

DC3

简单普及一下一种线性构造算法 DC3。

为什么叫 DC3 呢？

DC3

简单普及一下一种线性构造算法 DC3。

为什么叫 DC3 呢？因为它和 3 有很大关联。

DC3

简单普及一下一种线性构造算法 DC3。

为什么叫 DC3 呢？因为它和 3 有很大关联。

简单来说，算法分成三步。

DC3

简单普及一下一种线性构造算法 DC3。

为什么叫 DC3 呢？因为它和 3 有很大关联。

简单来说，算法分成三步。

- 所有后缀分为两部分：后缀起始位置 $\text{mod } 3 = 0$ 和后缀起始位置 $\text{mod } 3 \neq 0$ 。

DC3

简单普及一下一种线性构造算法 DC3。

为什么叫 DC3 呢？因为它和 3 有很大关联。

简单来说，算法分成三步。

- 所有后缀分为两部分：后缀起始位置 $\text{mod } 3 = 0$ 和后缀起始位置 $\text{mod } 3 \neq 0$ 。
- 第一部分递归求解，然后利用第一部分对于第二部分进行基数排序。

DC3

简单普及一下一种线性构造算法 DC3。

为什么叫 DC3 呢？因为它和 3 有很大关联。

简单来说，算法分成三步。

- 所有后缀分为两部分：后缀起始位置 $\text{mod } 3 = 0$ 和后缀起始位置 $\text{mod } 3 \neq 0$ 。
- 第一部分递归求解，然后利用第一部分对于第二部分进行基数排序。
- 最后讨论两种情况得到两个部分合并后的结果。

DC3

简单普及一下一种线性构造算法 DC3。

为什么叫 DC3 呢？因为它和 3 有很大关联。

简单来说，算法分成三步。

- 所有后缀分为两部分：后缀起始位置 $\text{mod } 3 = 0$ 和后缀起始位置 $\text{mod } 3 \neq 0$ 。
- 第一部分递归求解，然后利用第一部分对于第二部分进行基数排序。
- 最后讨论两种情况得到两个部分合并后的结果。

复杂度证明的话，不难发现瓶颈在于第一个部分的排序。

DC3

简单普及一下一种线性构造算法 DC3。

为什么叫 DC3 呢？因为它和 3 有很大关联。

简单来说，算法分成三步。

- 所有后缀分为两部分：后缀起始位置 $\text{mod } 3 = 0$ 和后缀起始位置 $\text{mod } 3 \neq 0$ 。
- 第一部分递归求解，然后利用第一部分对于第二部分进行基数排序。
- 最后讨论两种情况得到两个部分合并后的结果。

复杂度证明的话，不难发现瓶颈在于第一个部分的排序。


$$f(n) = O(n) + f(2n/3) \Rightarrow f(n) = O(n)$$

SA-IS

¹⁸<https://riteme.github.io/blog/2016-6-19/sais.html#21> ◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ ≡ ↺ 🔍 ↻

SA-IS

虽然 DC3 实际速度表现上还是要胜倍增一筹，但是它在常数以及空间上表现都不是那么的优秀，这时候你需要

¹⁸<https://riteme.github.io/blog/2016-6-19/sais.html#21> 

SA-IS

虽然 DC3 实际速度表现上还是要胜倍增一筹，但是它在常数以及空间上表现都不是那么的优秀，这时候你需要(DC1024)

¹⁸<https://riteme.github.io/blog/2016-6-19/sais.html#21>

SA-IS


虽然 DC3 实际速度表现上还是要胜倍增一筹，但是它在常数以及空间上表现都不是那么的优秀，这时候你需要(DC1024)SA-IS。

¹⁸<https://riteme.github.io/blog/2016-6-19/sais.html#21>

SA-IS

虽然 DC3 实际速度表现上还是要胜倍增一筹，但是它在常数以及空间上表现都不是那么的优秀，这时候你需要(DC1024)SA-IS。

同样这也是个利用递归的单层来进行复杂度证明为线性的做法。

¹⁸<https://riteme.github.io/blog/2016-6-19/sais.html#21> 

SA-IS

虽然 DC3 实际速度表现上还是要胜倍增一筹，但是它在常数以及空间上表现都不是那么的优秀，这时候你需要(DC1024)SA-IS。

同样这也是个利用递归的单层来进行复杂度证明为线性的做法。

但由于这个实现原理较为复杂，本处只提及，并不讲解。有一篇很写的很好的博客¹⁸。

¹⁸<https://riteme.github.io/blog/2016-6-19/sais.html#21>

Height

有了 ~~sa~~ 其实我们搞不了太大的新闻

我们就可以弄出一个 $height[i]$ ，其定义为 $sa[i-1]$ 和 $sa[i]$ 的最长公共前缀 (LCP) 的长度。

暴力求是 $O(n^2)$ 的，然后显然我们要考虑优化了。

二分哈希

二分哈希

同样求 LCP 我们可以用二分哈希优化到 $O(n \log n)$ 。

二分哈希

同样求 LCP 我们可以用二分哈希优化到 $O(n \log n)$ 。
但没有那么优美，而且常数也不小，显然有更好的方式。

性质优化

性质优化

我们考虑暴力匹配的时候，计算两个相邻的 LCP 的时候，没必要重新从 0 开始枚举。

性质优化

我们考虑暴力匹配的时候，计算两个相邻的 LCP 的时候，没必要重新从 0 开始枚举。

我们可以有如下一条性质。从 $sa[i-1]$ 推导到 $sa[i]$ 的时候， $height[i]$ 最多会减少 1。

性质优化

我们考虑暴力匹配的时候，计算两个相邻的 LCP 的时候，没必要重新从 0 开始枚举。
我们可以有如下一条性质。从 $sa[i-1]$ 推导到 $sa[i]$ 的时候， $height[i]$ 最多会减少 1。

证明：设排在后缀 $i-1$ 前一个的是后缀 k 。后缀 k 和后缀 $i-1$ 分别得到后缀 $k+1$ 和后缀 i ，因此后缀 $k+1$ 一定排在后缀 i ，并且最长公共前缀长度为 $height[rank[i-1]]-1$ 。

性质优化

我们考虑暴力匹配的时候，计算两个相邻的 LCP 的时候，没必要重新从 0 开始枚举。我们可以有如下一条性质。从 $sa[i-1]$ 推导到 $sa[i]$ 的时候， $height[i]$ 最多会减少 1。

证明：设排在后缀 $i-1$ 前一个的是后缀 k 。后缀 k 和后缀 $i-1$ 分别得到后缀 $k+1$ 和后缀 i ，因此后缀 $k+1$ 一定排在后缀 i ，并且最长公共前缀长度为 $height[rank[i-1]]-1$ 。

然后每次暴力继承前面，然后暴力比较即可。复杂度是 $O(n)$ 的，因为 $height$ 每次 -1 且最多到 $n-1$ 。

结论

结论

有 *height* 有什么用呢？主要是有这样一个结论。

结论

有 *height* 有什么用呢？主要是有这样一个结论。

对于两个后缀 j 和 k ，不妨设 $\text{rank}[j] < \text{rank}[k]$ ，则不难证明后缀 j 和 k 的 LCP 的长度等于

$$\min_{i=\text{rank}[j]+1}^{\text{rank}[k]} \text{height}[i]$$

然后求任意两个后缀的 LCP，我们利用 *ST* 表就可以优化这个求 min 的过程，做到 $O(n \log n) - O(1)$ 了。

结论

有 *height* 有什么用呢？主要是有这样一个结论。

对于两个后缀 j 和 k ，不妨设 $\text{rank}[j] < \text{rank}[k]$ ，则不难证明后缀 j 和 k 的 LCP 的长度等于

$$\min_{i=\text{rank}[j]+1}^{\text{rank}[k]} \text{height}[i]$$

然后求任意两个后缀的 LCP，我们利用 *ST* 表就可以优化这个求 min 的过程，做到 $O(n \log n) - O(1)$ 了。

闲着无聊也可以建笛卡尔树加分块 RMQ 可以做到 $O(n) - O(1)$ 。

[USACO2006 Dec]Milk Patterns 产奶的模式¹⁹

给你一个长为 n 的序列 A ，找出其中**最长**的一个子串满足它在 A 中**至少**出现 k 次。

$$1 \leq k \leq n \leq 20000$$

¹⁹BZOJ 1717

[USACO2006 Dec]Milk Patterns 产奶的模式

[USACO2006 Dec]Milk Patterns 产奶的模式

像这种至少出现 k 次求 \min or \max 的题，不难想到二分答案。

问题就转化为，是否存在一个 $\geq \lim$ 的子串至少出现 k 次。

[USACO2006 Dec]Milk Patterns 产奶的模式

像这种至少出现 k 次求 \min or \max 的题，不难想到二分答案。

问题就转化为，是否存在一个 $\geq \lim$ 的子串至少出现 k 次。

一个子串可以表示成两个后缀（可以相同）的公共前缀，所以转化成是否存在一段长度 $\geq k-1$ 连续的 $height \geq \lim$ 序列。

直接扫过去就行了，复杂度是 $O(n \log n)$ 的。

L-Gap Substrings²⁰

如果一个字符串能写成 UVU 的形式，其中 U 非空， V 恰好有 L 个字符，那么我们称这个字符串为 L -Gap 串。比如 abcbabc 就是一个 1-Gap 串。
给定字符串 S 和正整数 g ，统计 S 的 g -Gap 子串数量。

$$1 \leq |S| \leq 5 \times 10^4, 1 \leq g \leq 10$$

好像 g 的范围没有什么用，不要被误导了。

L-Gap Substrings

L-Gap Substrings

这是一道经典套路题。如果做过「NOI2016」优秀的拆分²¹ 的应该都会做。

²¹LOJ 2083

L-Gap Substrings

这是一道经典套路题。如果做过「NOI2016」优秀的拆分²¹ 的应该都会做。

考虑枚举 U 的长度 l 。那么我们把 S 分成 $\lceil \frac{|S|}{l} \rceil$ 块，其中第 i 块的起点为 $p = (l-1) \times i + 1$ 。我们求出 p 与 $p + g + l$ 为开头的最长公共前缀，和以结尾的最长公共后缀，他们的和也就是能同时扩展的最大长度 len 。

那么我们发现 U 在长为 len 的区间内滑动都是能对应到合法解的，把答案加上 $len - l + 1$ 。

²¹LOJ 2083

L-Gap Substrings

这是一道经典套路题。如果做过「NOI2016」优秀的拆分²¹ 的应该都会做。考虑枚举 U 的长度 l 。那么我们把 S 分成 $\lceil \frac{|S|}{l} \rceil$ 块，其中第 i 块的起点为 $p = (l-1) \times i + 1$ 。我们求出 p 与 $p + g + l$ 为开头的最长公共前缀，和以结尾的最长公共后缀，他们的和也就是能同时扩展的最大长度 len 。那么我们发现 U 在长为 len 的区间内滑动都是能对应到合法解的，把答案加上 $len - l + 1$ 。至于正确性？首先考虑长为 l 的 U 一定会跨区间，而且位置需要隔着 g 刚好对应。总区间个数是调和级数只有 $|S| \ln |S|$ 个。所以最后复杂度就是 $O(|S| \log |S|)$ 的。

²¹LOJ 2083

[TJOI2016 & HEOI2016] 字符串²²

一个长为 n 的字符串 s ，和 m 个询问。每次询问有 4 个参数分别为 a, b, c, d 。要你告诉它 $s[a...b]$ 中的所有子串和 $s[c...d]$ 的最长公共前缀 (LCP) 的最大值。

$$1 \leq n, m \leq 10^5, a \leq b, c \leq d, 1 \leq a, b, c, d \leq n$$

[TJOI2016 & HEOI2016] 字符串

[TJOI2016 & HEOI2016] 字符串

首先我们考虑与 $[c, d]$ 有最长 LCP 的在哪里。不难发现，就是后缀排序后 $rk[i]$ 与 $rk[c]$ 最靠近的 i 。

[TJOI2016 & HEOI2016] 字符串

首先我们考虑与 $[c, d]$ 有最长 LCP 的在哪里。不难发现，就是后缀排序后 $rk[i]$ 与 $rk[c]$ 最靠近的 i 。

答案表示出来大概是这样的。

$$\text{ans} = \min(d - c + 1, \max_{i=a}^b \{\min(\text{LCP}(i, c), b - i + 1)\})$$

[TJOI2016 & HEOI2016] 字符串

首先我们考虑与 $[c, d]$ 有最长 LCP 的在哪里。不难发现，就是后缀排序后 $rk[i]$ 与 $rk[c]$ 最靠近的 i 。

答案表示出来大概是这样的。

$$\text{ans} = \min(d - c + 1, \max_{i=a}^b \{\min(\text{LCP}(i, c), b - i + 1)\})$$

我们发现直接求这个 i 会被后面的 $b - i + 1$ 限制掉，所以不能直接这样求。

[TJOI2016 & HEOI2016] 字符串

首先我们考虑与 $[c, d]$ 有最长 LCP 的在哪里。不难发现，就是后缀排序后 $rk[i]$ 与 $rk[c]$ 最靠近的 i 。

答案表示出来大概是这样的。

$$\text{ans} = \min(d - c + 1, \max_{i=a}^b \{\min(\text{LCP}(i, c), b - i + 1)\})$$

我们发现直接求这个 i 会被后面的 $b - i + 1$ 限制掉，所以不能直接这样求。

但我们可以考虑转化一下我们考虑 **二分答案** 如果判断是否存在就容易一些。

我们考虑二分这个长度假设是 len 那么前面的 i 就只能存在于 $[a, b - len + 1]$ 这个区间内。

然后看 $rk[c]$ 周围连续的 $height[q] \geq len$ 可以延伸到哪个范围，利用 **倍增** 就可以实现。

得到这个区间 $[sl, sr]$ 后我们就需要查找里面是否存在 $[a, b - len + 1]$ 的元素这个东西就直接上 **主席树**就行了。

[TJOI2016 & HEOI2016] 字符串

首先我们考虑与 $[c, d]$ 有最长 LCP 的在哪里。不难发现，就是后缀排序后 $rk[i]$ 与 $rk[c]$ 最靠近的 i 。

答案表示出来大概是这样的。

$$\text{ans} = \min(d - c + 1, \max_{i=a}^b \{\min(\text{LCP}(i, c), b - i + 1)\})$$

我们发现直接求这个 i 会被后面的 $b - i + 1$ 限制掉，所以不能直接这样求。

但我们可以考虑转化一下我们考虑 **二分答案** 如果判断是否存在就容易一些。

我们考虑二分这个长度假设是 len 那么前面的 i 就只能存在于 $[a, b - len + 1]$ 这个区间内。

然后看 $rk[c]$ 周围连续的 $height[q] \geq len$ 可以延伸到哪个范围，利用 **倍增** 就可以实现。

得到这个区间 $[sl, sr]$ 后我们就需要查找里面是否存在 $[a, b - len + 1]$ 的元素这个东西就直接上 **主席树**就行了。

复杂度是 $O(n \log n)$ 的，常数有点大，还很不好写。。

[TJOI2016 & HEOI2016] 字符串

[TJOI2016 & HEOI2016] 字符串

然后本人写完后常数过大，怎么卡都过不了 BZOJ。。。后来翻了别人代码，又短又快！后来发现是暴力。。。

[TJOI2016 & HEOI2016] 字符串

然后本人写完后常数过大，怎么卡都过不了 BZOJ。。。后来翻了别人代码，又短又快！后来发现是暴力。。。

我们继续考虑之前答案的那个式子

$$\text{ans} = \min(d - c + 1, \max_{i=a}^b \{\min(\text{LCP}(i, c), b - i + 1)\})$$

我们考虑最初的想法，向 $rk[c]$ 前后去扫一下得到答案，其中如果此处 $sa[i]$ 在 $[a, b]$ 之中的话我们就计入答案就行了。

如果 $\min \text{height} \leq \text{ans}$ 那么我们就退出即可。

[TJOI2016 & HEOI2016] 字符串

然后本人写完后常数过大，怎么卡都过不了 BZOJ。。。后来翻了别人代码，又短又快！后来发现是暴力。。。

我们继续考虑之前答案的那个式子

$$\text{ans} = \min(d - c + 1, \max_{i=a}^b \{\min(\text{LCP}(i, c), b - i + 1)\})$$

我们考虑最初的想法，向 $rk[c]$ 前后去扫一下得到答案，其中如果此处 $sa[i]$ 在 $[a, b]$ 之中的话我们就计入答案就行了。

如果 $\min height \leq ans$ 那么我们就退出即可。

虽然理论复杂度是 $O(nq)$ 的。但这样在大部分不是特意构造的数据下，十分的优秀。

Suffix Automaton

²³<https://www.cnblogs.com/zjp-shadow/p/9218214.html>

Suffix Automaton

Suffix Automaton 俗称 SAM，是解决大部分字符串题的绝佳利器。

²³<https://www.cnblogs.com/zjp-shadow/p/9218214.html>

Suffix Automaton

Suffix Automaton 俗称 SAM，是解决大部分字符串题的绝佳利器。

对于一个字符串 S ，它对应的 SAM 是一个**最小**的 DFA 满足接受且只接受 S 的后缀。

DAWG 和它的唯一区别在于，DAWG 把每个状态都视为终态集的元素，也就是意味着可以接受 S 的所有子串。

由于时间与篇幅问题，此处只简单讲下大概原理以及构造流程，具体原因以及证明可以参考此处²³。

²³<https://www.cnblogs.com/zjp-shadow/p/9218214.html>

Concept

Concept

首先我们先介绍一个概念 **子串的结束位置集合** $endpos(right)$ 。

对于 S 的一个子串 s ， $endpos(s) = s$ 在 S 中所有出现的结束位置集合。

SAM 中的一个状态包含的子串都具有相同的 $endpos$ ，它们都互为后缀。

Concept

首先我们先介绍一个概念 **子串结束位置集合** $endpos(right)$ 。

对于 S 的一个子串 s ， $endpos(s) = s$ 在 S 中所有出现的结束位置集合。

SAM 中的一个状态包含的子串都具有相同的 $endpos$ ，它们都互为后缀。

然后我们发现对于一个连续的后缀会在 SAM 被断开，我们用后缀连接 $link$ 把这些断开的状态连起来，类似于 AC 自动机中的 $fail$ 。

然后一条连续的 $fail$ 跳的路径，我们称作 $suffix-path$ 。

Concept

首先我们先介绍一个概念 **子串的结束位置集合** $endpos(right)$ 。

对于 S 的一个子串 s ， $endpos(s) = s$ 在 S 中所有出现的结束位置集合。

SAM 中的一个状态包含的子串都具有相同的 $endpos$ ，它们都互为后缀。

然后我们发现对于一个连续的后缀会在 SAM 被断开，我们用后缀连接 $link$ 把这些断开的状态连起来，类似于 AC 自动机中的 $fail$ 。

然后一条连续的 $fail$ 跳的路径，我们称作 $suffix-path$ 。

还要处理出每个状态 st 包含的最长子串的长度 $maxlen[st]$ 。

Construction Method

Construction Method

通常实现的方法都是增量法，因为这种方法可以支持动态插入，复杂度也是线性的。

Construction Method

通常实现的方法都是增量法，因为这种方法可以支持动态插入，复杂度也是线性的。

考虑当前添加的为 $S[i+1]$ ，我们对于这个节点新建状态 v 。那么 $\forall j \in [1, i], S[j...i]$ 的状态 p 都要添加一个 $\delta(p, S[i]) = v$ 。如果还没有 $\delta(p, S[i])$ ，那么可以直接添加，然后使得 $p \rightarrow \text{link}[p]$ ，也就是沿着 *suffix-path* 走。

Construction Method

通常实现的方法都是增量法，因为这种方法可以支持动态插入，复杂度也是线性的。

考虑当前添加的为 $S[i+1]$ ，我们对于这个节点新建状态 v 。那么 $\forall j \in [1, i], S[j...i]$ 的状态 p 都要添加一个 $\delta(p, S[i]) = v$ 。如果还没有 $\delta(p, S[i])$ ，那么可以直接添加，然后使得 $p \rightarrow \text{link}[p]$ ，也就是沿着 *suffix-path* 走。

如果 p 跳到了 q_0 ，那么令 $\text{link}[v] = q_0$ 即可。（ q_0 只有一个节点可以不看做集合）

否则我们令 $p \leftarrow \delta(p, S[i])$ ，然后分以下两种情况讨论。

Construction Method

通常实现的方法都是增量法，因为这种方法可以支持动态插入，复杂度也是线性的。

考虑当前添加的为 $S[i+1]$ ，我们对于这个节点新建状态 v 。那么 $\forall j \in [1, i], S[j...i]$ 的状态 p 都要添加一个 $\delta(p, S[i]) = v$ 。如果还没有 $\delta(p, S[i])$ ，那么可以直接添加，然后使得 $p \rightarrow \text{link}[p]$ ，也就是沿着 *suffix-path* 走。

如果 p 跳到了 q_0 ，那么令 $\text{link}[v] = q_0$ 即可。（ q_0 只有一个节点可以不看做集合）

否则我们令 $p \leftarrow \delta(p, S[i])$ ，然后分以下两种情况讨论。

- 如果 $\text{maxlen}[q] = \text{maxlen}[p] + 1$ ，直接令 $\text{link}[v] = q_0$ 。

Construction Method

通常实现的方法都是增量法，因为这种方法可以支持动态插入，复杂度也是线性的。

考虑当前添加的为 $S[i+1]$ ，我们对于这个节点新建状态 v 。那么 $\forall j \in [1, i], S[j...i]$ 的状态 p 都要添加一个 $\delta(p, S[i]) = v$ 。如果还没有 $\delta(p, S[i])$ ，那么可以直接添加，然后使得 $p \rightarrow \text{link}[p]$ ，也就是沿着 *suffix-path* 走。

如果 p 跳到了 q_0 ，那么令 $\text{link}[v] = q_0$ 即可。（ q_0 只有一个节点可以不看做集合）

否则我们令 $p \leftarrow \delta(p, S[i])$ ，然后分以下两种情况讨论。

- 如果 $\text{maxlen}[q] = \text{maxlen}[p] + 1$ ，直接令 $\text{link}[v] = q_0$ 。
- 否则需要 q 拆成两个状态 clone, q ，然后令 $\text{maxlen}[\text{clone}] \leftarrow \text{maxlen}[p] + 1$ 使得它满足前面的条件。然后把在 *suffix-path* 上满足 $\delta(p, S[i]) = q$ 的变为 clone 。最后把 $\text{link}[v], \text{link}[q]$ 都变为 clone 。

Construction Method

通常实现的方法都是增量法，因为这种方法可以支持动态插入，复杂度也是线性的。

考虑当前添加的为 $S[i+1]$ ，我们对于这个节点新建状态 v 。那么 $\forall j \in [1, i], S[j...i]$ 的状态 p 都要添加一个 $\delta(p, S[i]) = v$ 。如果还没有 $\delta(p, S[i])$ ，那么可以直接添加，然后使得 $p \rightarrow \text{link}[p]$ ，也就是沿着 *suffix-path* 走。

如果 p 跳到了 q_0 ，那么令 $\text{link}[v] = q_0$ 即可。（ q_0 只有一个节点可以不看做集合）

否则我们令 $p \leftarrow \delta(p, S[i])$ ，然后分以下两种情况讨论。

- 如果 $\text{maxlen}[q] = \text{maxlen}[p] + 1$ ，直接令 $\text{link}[v] = q_0$ 。
- 否则需要 q 拆成两个状态 clone, q ，然后令 $\text{maxlen}[\text{clone}] \leftarrow \text{maxlen}[p] + 1$ 使得它满足前面的条件。然后把在 *suffix-path* 上满足 $\delta(p, S[i]) = q$ 的变为 clone 。最后把 $\text{link}[v], \text{link}[q]$ 都变为 clone 。

这样就可以做完了，状态最多是 $2n - 1$ ，转移最多是 $3n - 4$ 的 ($n \geq 3$)。所以记得要开两倍空间。

后缀自动机四 · 重复旋律 ²⁴

给你 n 个仅由数字字符构成的字符串 S 。

统计其中本质不同的子串代表的数之和（允许有前导 0）。

答案对于 $10^9 + 7$ 取模。 $\sum |S| \leq 10^6$

后缀自动机四·重复旋律 7

后缀自动机四 · 重复旋律 7

此题就是要统计所有本质不同的子串权值和，对于每个子串权值定义就是它在十进制下的值。因为每个数都是从前往后构成的，并且 *SAM* 上每个状态的 *substrings* 是从起点开始的路径构成的单词集合。

又由于正向的转移函数 $trans[u][1..c]$ 是一个 *DAG*。

不难考虑用正向拓扑 *dp* 求解这个值。令 dp_i 为状态 i 所有 *substrings*(i) 的权值和，那

么显然有 $dp_v = \sum_{trans[u][id]} dp[u] \times 10 + id$ 。但这样显然会错...

后缀自动机四 · 重复旋律 7

此题就是要统计所有本质不同的子串权值和，对于每个子串权值定义就是它在十进制下的值。因为每个数都是从前往后构成的，并且 SAM 上每个状态的 *substrings* 是从起点开始的路径构成的单词集合。

又由于正向的转移函数 $trans[u][1..c]$ 是一个 DAG。

不难考虑用正向拓扑 dp 求解这个值。令 dp_i 为状态 i 所有 *substrings*(i) 的权值和，那

么显然有 $dp_v = \sum_{trans[u][id]} dp[u] \times 10 + id$ 。但这样显然会错...

因为一个状态可能有很多子串加上了 id 这个值，但我们只加上了一个，所以我们记下每个状态具有的子串个数 tot_i 。那么有 $tot_v = \sum_{trans[u][id]} tot_u$ 。又有

$$dp_v = \sum_{trans[u][id]} dp[u] \times 10 + id \times tot_u。$$

后缀自动机四 · 重复旋律 7

此题就是要统计所有本质不同的子串权值和，对于每个子串权值定义就是它在十进制下的值。因为每个数都是从前往后构成的，并且 SAM 上每个状态的 *substrings* 是从起点开始的路径构成的单词集合。

又由于正向的转移函数 $\text{trans}[u][1..c]$ 是一个 DAG。

不难考虑用正向拓扑 dp 求解这个值。令 dp_i 为状态 i 所有 *substrings*(i) 的权值和，那么显然有 $dp_v = \sum_{\text{trans}[u][id]} dp[u] \times 10 + id$ 。但这样显然会错...

$$dp_v = \sum_{\text{trans}[u][id]} dp[u] \times 10 + id$$

因为一个状态可能有很多子串加上了 id 这个值，但我们只加上了一个，所以我们记下每个状态具有的子串个数 tot_i 。那么有 $tot_v = \sum_{\text{trans}[u][id]} tot_u$ 。又有

$$dp_v = \sum_{\text{trans}[u][id]} dp[u] \times 10 + id \times tot_u$$

但是这个是有许多串一起询问答案，可以用 **广义后缀自动机**来解决。

其实这题我们可以用当初做后缀数组题的一些思想，我们对于许多子串在中间加入一些字符例如 \vdash 将其隔开，然后每次统计的时候不能统计中间具有 \vdash 的字符串，对于这些枚举的边为这些转移的，我们就不转移 dp, tot 就可以了。

「NOI2018」你的名字²⁵

给你一个母串 S 。

有 Q 次询问，每次给一个串 T 和 l, r ，询问 T 有多少个本质不同的子串在 $S_{l,r}$ 中没有出现。

$$|S|, |T| \leq 5 \times 10^5, \sum |T| \leq 10^6$$

对于 68pts 有 $l = 1, r = n$ 。

「NOI2018」你的名字

²⁶主席树写法根据常数有 40 ~ 100 分，暴力直接就有 68pts

「NOI2018」你的名字

不出现显然是不好做的，我们可以补集转化考虑出现的本质不同的串。

²⁶主席树写法根据常数有 40 ~ 100 分，暴力直接就有 68pts

「NOI2018」你的名字

不出现显然是不好做的，我们可以补集转化考虑出现的本质不同的串。

然后是考虑子串，显然不好考虑。我们可以枚举 T 的每个后缀/前缀，然后看它和 $S_{l,r}$ 的 LCP/LCS 的长度，就能算出有多少个公共子串了。

²⁶主席树写法根据常数有 40 ~ 100 分，暴力直接就有 68pts

「NOI2018」你的名字

不出现显然是不好做的，我们可以补集转化考虑出现的本质不同的串。

然后是考虑子串，显然不好考虑。我们可以枚举 T 的每个后缀/前缀，然后看它和 $S_{l,r}$ 的 LCP/LCS 的长度，就能算出有多少个公共子串了。

这样转化后和前面那道 **[TJOI2016 & HEOI2016] 字符串** 就一模一样了。但直接实现是 $O(\sum |T| \log^2(|S| + |T|))$ 的，而且数据造的很强，暴力过不去²⁶。

²⁶主席树写法根据常数有 40 ~ 100 分，暴力直接就有 68pts

「NOI2018」你的名字

「NOI2018」你的名字

但显然用 SAM 可以更好地解决这题。

首先考虑前 68pts。我们把 T 在 S 上不断匹配，如果不存在转移那么就跳 $Link$ 。然后对于 T 第 i 位匹配上的长度就是 T 前 i 个字母构成的前缀与 S 的 LCS 长度。

「NOI2018」你的名字

但显然用 SAM 可以更好地解决这题。

首先考虑前 68pts。我们把 T 在 S 上不断匹配，如果不存在转移那么就跳 $Link$ 。然后对于 T 第 i 位匹配上的长度就是 T 前 i 个字母构成的前缀与 S 的 LCS 长度。

如果我们有区间 $[l, r]$ 的限制如何解决呢？我们同样可以在 SAM 上遍历，假设当前匹配到位置 u 的长度是 $len(len \leq maxlen_u)$ 。

「NOI2018」你的名字

但显然用 SAM 可以更好地解决这题。

首先考虑前 68pts。我们把 T 在 S 上不断匹配，如果不存在转移那么就跳 $Link$ 。然后对于 T 第 i 位匹配上的长度就是 T 前 i 个字母构成的前缀与 S 的 LCS 长度。

如果我们有区间 $[l, r]$ 的限制如何解决呢？我们同样可以在 SAM 上遍历，假设当前匹配到位置 u 的长度是 $len (len \leq maxlen_u)$ 。

那么对于 $fail$ 树上 u 的子树需要有对应到 S 串上 $[l + len - 1, r]$ 的节点。那么一开始直接主席树合并 $fail$ 树，然后就可以实现判断过程了。

注意当 $len = maxlen_{link_u}$ 的时候需要跳 $link$ 。

渐进复杂度是 $O(|S| \log |S| + \sum |T| \log |S|)$ 的。

「九省联考 2018」制胡闹²⁷

给定一个长度为 n 的仅由数字构成的字符串 S 。

现在有 q 次询问，会给出 S 的一个字符串 $S_{l,r}$ ，请你求出有多少对 (i, j) ，满足 $1 \leq i < j \leq n$ ， $i+1 < j$ ，且 $S_{l,r}$ 出现在 $S_{1,i}$ 中或 $S_{i+1,j-1}$ 中或 $S_{j,n}$ 中。

$$n \leq 10^5, q \leq 3 \times 10^5$$

²⁷LOJ 2479

「九省联考 2018」制胡闹

「九省联考 2018」制胡闹

这种几个要求任一满足的计数通常都可以暴力容斥转化。

那么就是计算至少在一个区间的答案减去至少在两个区间的答案加上三个区间都有的答案。

「九省联考 2018」制胡闹

这种几个要求任一满足的计数通常都可以暴力容斥转化。

那么就是计算至少在一个区间的答案减去至少在两个区间的答案加上三个区间都有的答案。

然后就可以开始愉快的分类讨论啦 QwQ 。

「九省联考 2018」制胡闹

这种几个要求任一满足的计数通常都可以暴力容斥转化。

那么就是计算至少在一个区间的答案减去至少在两个区间的答案加上三个区间都有的答案。

然后就可以开始愉快的分类讨论啦 QwQ。

我们先考虑哪些位置可以作为结束的端点，其实就是 SAM 上对应 $S_{l,r}$ 节点的 $fail$ 子树的所有 $endpos$ 集合。

「九省联考 2018」制胡闹

这种几个要求任一满足的计数通常都可以暴力容斥转化。

那么就是计算至少在一个区间的答案减去至少在两个区间的答案加上三个区间都有的答案。

然后就可以开始愉快的分类讨论啦 QwQ。

我们先考虑哪些位置可以作为结束的端点，其实就是 SAM 上对应 $S_{l,r}$ 节点的 *fail* 子树的所有 *endpos* 集合。

如何找这个合法节点呢？我们可以在 S_r 对应的节点处倍增跳 *fail* 树的祖先满足 \maxlen 不小于 $r-l+1$ 最高的点即可。

「九省联考 2018」制胡闹

这种几个要求任一满足的计数通常都可以暴力容斥转化。

那么就是计算至少在一个区间的答案减去至少在两个区间的答案加上三个区间都有的答案。

然后就可以开始愉快的分类讨论啦 QwQ。

我们先考虑哪些位置可以作为结束的端点，其实就是 SAM 上对应 $S_{l,r}$ 节点的 *fail* 子树的所有 *endpos* 集合。

如何找这个合法节点呢？我们可以在 S_r 对应的节点处倍增跳 *fail* 树的祖先满足 $maxlen$ 不小于 $r-l+1$ 最高的点即可。

然后就可以开始对于这些进行讨论，需要支持查询区间最大最小值、相邻两点的贡献之和、区间上二分的线段树。离线后进行线段树合并讨论贡献即可。

由于细节过于繁琐可以自行找网上博客查看讲解。

Thanks