哈希、前缀函数、KMP、Z 函数

宋佳兴

2024 年 7 月 22 日

- 欢迎有想法的同学随时上来与我交流。
- 有任何问题，请随时提问。
- 讲课过程中途，我们将会休息 15 到 20 分钟。

CF126B Password

给定一个字符串 s，需要找到一个最长的子串 t，使得 t 同时是 s 的前缀和后缀，且 t 在 s 的中间部分也出现过（即 t 不是 s 的开头或结尾）。

$n \leq 10^6$

题目链接

CF126B Password

设 pre_k 表示长度为 k 的前缀，答案必须是某个 pre_k。

CF126B Password

设 pre_k 表示长度为 k 的前缀，答案必须是某个 pre_k。

pre_k 合法需要满足两个条件：

- pre_k 是 s 的 border，通过 fail 数组可以轻松判定。

CF126B Password

设 pre_k 表示长度为 k 的前缀，答案必须是某个 pre_k。

pre_k 合法需要满足两个条件：

- pre_k 是 s 的 border，通过 fail 数组可以轻松判定。
- pre_k 是 s 的**真前缀**的 border，即存在 $i \le n-1$ 满足 $fail_i = k$。

CF126B Password

设 pre_k 表示长度为 k 的前缀，答案必须是某个 pre_k。

pre_k 合法需要满足两个条件：

- pre_k 是 s 的 border，通过 fail 数组可以轻松判定。
- pre_k 是 s 的**真前缀**的 border，即存在 $i \le n-1$ 满足 $fail_i = k$。

预处理 fail 数组后，就能在 $O(1)$ 时间内判断一个 pre_k 是否合法，然后取最大的 k 作为答案。

CF898F Restoring the Expression

给定一个由数字组成的非空字符串 s，你需要在该字符串中插入一个字符'$+$' 和一个字符'$=$' 使得形成一个表达式 $a+b=c$，其中 a,b,c 是非负整数且没有前导零，并且表达式 $a+b=c$ 成立。题目保证答案是存在的。

$n \leq 10^6$

提示：

题目链接

CF898F Restoring the Expression

给定一个由数字组成的非空字符串 s，你需要在该字符串中插入一个字符'$+$' 和一个字符'$=$' 使得形成一个表达式 $a+b=c$，其中 a,b,c 是非负整数且没有前导零，并且表达式 $a+b=c$ 成立。题目保证答案是存在的。

$n \le 10^6$

提示：使用哈希方法。

题目链接

CF898F Restoring the Expression

朴素做法是枚举'+' 和'=' 的位置，并 $O(n)$ 判断 $a+b=c$ 是否成立，复杂度为 $O(n^3)$。

CF898F Restoring the Expression

朴素做法是枚举'+' 和'=' 的位置，并 $O(n)$ 判断 $a+b=c$ 是否成立，复杂度为 $O(n^3)$。

能否更快地检验 $a+b=c$? 我们发现哈希可以做到。设 $base=10$，哈希能在 $O(1)$ 时间内检验等式，复杂度降为 $O(n^2)$。

CF898F Restoring the Expression

朴素做法是枚举'+' 和'=' 的位置，并 $O(n)$ 判断 $a+b=c$ 是否成立，复杂度为 $O(n^3)$。

能否更快地检验 $a+b=c$? 我们发现哈希可以做到。设 $base=10$，哈希能在 $O(1)$ 时间内检验等式，复杂度降为 $O(n^2)$。

进一步，我们可以将'+' 和'=' 的枚举降为 $O(n)$。

CF898F Restoring the Expression

朴素做法是枚举'+' 和'=' 的位置，并 $O(n)$ 判断 $a+b=c$ 是否成立，复杂度为 $O(n^3)$。

能否更快地检验 $a+b=c$？我们发现哈希可以做到。设 $base=10$，哈希能在 $O(1)$ 时间内检验等式，复杂度降为 $O(n^2)$。

进一步，我们可以将'+' 和'=' 的枚举降为 $O(n)$。

我们发现 a, b, c 的位数之间存在关系，即 $len(c)=\max(len(a), len(b))+(0 \text{ or } 1)$，因此枚举量也降为 $O(1)$。

Z 函数（又名扩展 KMP）

定义：对于一个长度为 n 的字符串 s（下标从 0 开始），定义函数 z_i 表示 s 和 $s[i, n-1]$ 的最长公共前缀长度（LCP）。z 被称为 s 的 Z 函数，特别地，z_0 无意义。

来源：OI Wiki, Z 函数

Z 函数（又名扩展 KMP）

定义：对于一个长度为 n 的字符串 s（下标从 0 开始），定义函数 z_i 表示 s 和 $s[i, n-1]$ 的最长公共前缀长度（LCP）。z 被称为 s 的 Z 函数，特别地，z_0 无意义。

我们从 1 到 $n-1$ 依次计算 z_i 的值。在计算 z_i 的过程中，我们会利用已经计算好的 $z_1, \ldots, z_{i-1}$。

来源：OI Wiki，Z 函数

Z 函数（又名扩展 KMP）

定义：对于一个长度为 n 的字符串 s（下标从 0 开始），定义函数 z_i 表示 s 和 $s[i, n-1]$ 的最长公共前缀长度（LCP）。z 被称为 s 的 Z 函数，特别地，z_0 无意义。

我们从 1 到 $n-1$ 依次计算 z_i 的值。在计算 z_i 的过程中，我们会利用已经计算好的 $z_1, \ldots, z_{i-1}$。

对于 i，我们称区间 $[i, i+z_i-1]$ 为 i 的**匹配段**。

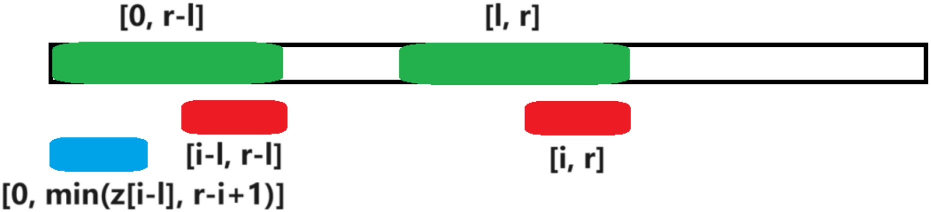
算法中我们维护已求出的匹配段中右端点最靠右的一个，记作 $[l, r]$。根据定义，$s[l, r]$ 是 s 的前缀。初始时 $l = r = -1$。

来源：OI Wiki, Z 函数

Z 函数

在计算 z_i 的过程中：

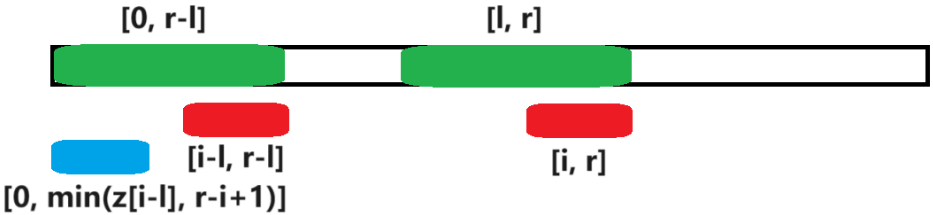
- 如果 $i \le r$，那么根据 $[l, r]$ 的定义有 $s[i, r] = s[i - l, r - l]$，因此 $z_i \ge \min(z_{i-l}, r - i + 1)$。我们令 $z_i = \min(z_{i-l}, r - i + 1)$，然后暴力枚举下一个字符，直到不能扩展为止。



Z 函数

在计算 z_i 的过程中:

- 如果 $i \le r$，那么根据 $[l, r]$ 的定义有 $s[i, r] = s[i - l, r - l]$，因此 $z_i \ge \min(z_{i-l}, r - i + 1)$。我们令 $z_i = \min(z_{i-l}, r - i + 1)$，然后暴力枚举下一个字符，直到不能扩展为止。

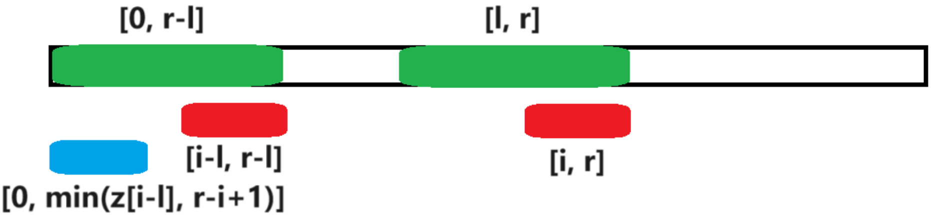


- 如果 $i > r$，那么我们直接暴力求出 z_i。

Z 函数

在计算 z_i 的过程中：

- 如果 $i \le r$，那么根据 $[l, r]$ 的定义有 $s[i, r] = s[i-l, r-l]$，因此 $z_i \ge \min(z_{i-l}, r-i+1)$。我们令 $z_i = \min(z_{i-l}, r-i+1)$，然后暴力枚举下一个字符，直到不能扩展为止。



- 如果 $i > r$，那么我们直接暴力求出 z_i。
- 在求出 z_i 后，使用 $[i, i+z_i-1]$ 更新 $[l, r]$。

Z 函数

```
int l = -1, r = -1;
for (int i = 1; i < n; i++) {
    int& j = z[i] = i <= r ? min(z[i - l], r - i + 1) : 0;
    while (s[i + j] == s[j]) j++;
    if (i + j > r) l = i, r = i + j - 1;
}
```

Z 函数

```
int l = -1, r = -1;
for (int i = 1; i < n; i++) {
    int& j = z[i] = i <= r ? min(z[i - l], r - i + 1) : 0;
    while (s[i + j] == s[j]) j++;
    if (i + j > r) l = i, r = i + j - 1;
}
```

复杂度分析：复杂度不明确的部分是 while 循环次数，我们分两种情况讨论：

Z 函数

```
int l = -1, r = -1;
for (int i = 1; i < n; i++) {
    int& j = z[i] = i <= r ? min(z[i - l], r - i + 1) : 0;
    while (s[i + j] == s[j]) j++;
    if (i + j > r) l = i, r = i + j - 1;
}
```

复杂度分析：复杂度不明确的部分是 while 循环次数，我们分两种情况讨论：

- 若 $z_{i-l} < r - i + 1$，则 $z_i = z_{i-l}$，while 会立刻跳出。

Z 函数

```cpp
int l = -1, r = -1;
for (int i = 1; i < n; i++) {
    int& j = z[i] = i <= r ? min(z[i - l], r - i + 1) : 0;
    while (s[i + j] == s[j]) j++;
    if (i + j > r) l = i, r = i + j - 1;
}
```

复杂度分析：复杂度不明确的部分是 while 循环次数，我们分两种情况讨论：

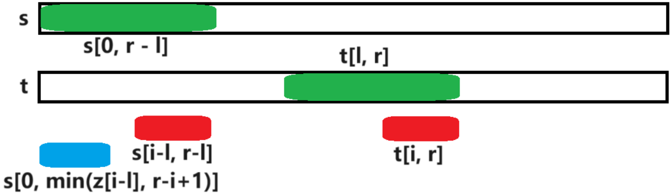
- 若 $z_{i-l} < r - i + 1$，则 $z_i = z_{i-l}$，while 会立刻跳出。
- 若 $z_{i-l} \geq r - i + 1$ 或 $i > r$，则 z_i 会初始化为 $r - i + 1$，此时 $i + z_i - 1 = r$，于是 while 每次执行都会使 r 后移一位，而 $r \leq n - 1$，所以总共执行 $O(n)$ 次。

Z 函数

匹配文本串：对于字符串 s 和 t，定义 zt_i 表示 s 和 $t[i, n-1]$ 的 LCP 长度，求 zt。

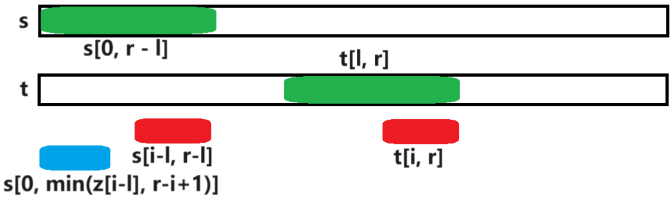
Z 函数

匹配文本串：对于字符串 s 和 t，定义 zt_i 表示 s 和 $t[i, n-1]$ 的 LCP 长度，求 zt。



Z 函数

匹配文本串：对于字符串 s 和 t，定义 zt_i 表示 s 和 $t[i, n-1]$ 的 LCP 长度，求 zt。




```
int l = -1, r = -1;
for (int i = 0; i < m; i++) {
    int& j = zt[i] = i <= r ? min(z[i - l], r - i + 1) : 0;
    while (j < n && t[i + j] == s[j]) j++;
    if (i + j > r) l = i, r = i + j - 1;
}
```

CF30E Tricky and Clever Password

给定一个字符串 s，将 s 分解为 $A + \text{prefix} + B + \text{middle} + C + \text{suffix}$ 满足：

- A, B, C 为任意字符串（可以为空）。
- prefix 和 suffix 互为反串（可以为空）。
- middle 是长度为奇数的回文串。

最大化 $\text{prefix} + \text{middle} + \text{suffix}$ 的长度。

$n \le 10^5$

题目链接

CF30E Tricky and Clever Password

可以证明 middle 一定是以某个点为中心的极长回文串，否则可以加长 middle，缩短 prefix 和 suffix 并保持不劣。

CF30E Tricky and Clever Password

可以证明 middle 一定是以某个点为中心的极长回文串，否则可以加长 middle，缩短 prefix 和 suffix 并保持不劣。

因此先枚举以 i 为中心的极长回文串 $[l, r]$，接下来的问题为：

- 找到最大的 k，使得 $s[n-k+1, n]$ 的反串为 $s[1, l-1]$ 的子串。

提示：

CF30E Tricky and Clever Password

可以证明 middle 一定是以某个点为中心的极长回文串，否则可以加长 middle，缩短 prefix 和 suffix 并保持不劣。

因此先枚举以 i 为中心的极长回文串 $[l, r]$，接下来的问题为：

- 找到最大的 k，使得 $s[n-k+1, n]$ 的反串为 $s[1, l-1]$ 的子串。

提示：二分答案。

CF30E Tricky and Clever Password

可以证明 middle 一定是以某个点为中心的极长回文串，否则可以加长 middle，缩短 prefix 和 suffix 并保持不劣。

因此先枚举以 i 为中心的极长回文串 $[l, r]$，接下来的问题为：

- 找到最大的 k，使得 $s[n-k+1, n]$ 的反串为 $s[1, l-1]$ 的子串。

提示：二分答案。

先通过两遍 Z 算法预处理 z_i 表示 LCP$(s[i, n], rev(s))$，二分答案为 k，则判定方法为 $\max z_{1..l-k} \geq k$，预处理前缀 max 即可。

CF30E Tricky and Clever Password

可以证明 middle 一定是以某个点为中心的极长回文串，否则可以加长 middle，缩短 prefix 和 suffix 并保持不劣。

因此先枚举以 i 为中心的极长回文串 $[l, r]$，接下来的问题为：

- 找到最大的 k，使得 $s[n-k+1, n]$ 的反串为 $s[1, l-1]$ 的子串。

提示：二分答案。

先通过两遍 Z 算法预处理 z_i 表示 LCP$(s[i, n], rev(s))$，二分答案为 k，则判定方法为 $\max z_{1..l-k} \geq k$，预处理前缀 max 即可。

复杂度 $O(n \log n)$，思考：怎么做到 $O(n)$。

NOIP2020 字符串匹配

小 C 需要找到字符串 S 的所有具有下列形式的拆分方案数：求 $S=(AB)^iC$ 的方案数，其中 $F(A)\le F(C)$，$F(S)$ 表示字符串 S 中出现奇数次的字符的数量，$(AB)^i$ 表示 AB 重复 i 遍。两种方案不同当且仅当拆分出的 A、B、C 中有至少一个字符串不同。

要求复杂度 $O(|S|)$。

提示：可以用到 Z 函数。

题目链接

NOIP2020 字符串匹配

注意到 AB 是 S 的前缀，因此第一个想法是枚举 AB 的长度和 i，然后计算 AB 有多少个前缀 A 满足 $F(A) \leq F(C)$。

NOIP2020 字符串匹配

注意到 AB 是 S 的前缀，因此第一个想法是枚举 AB 的长度和 i，然后计算 AB 有多少个前缀 A 满足 $F(A) \le F(C)$。

具体地，需要维护 cnt_k 表示当前有多少个前缀 A 满足 $F(A) = k$，利用哈希判定 $(AB)^i$ 是否是 S 的前缀，对每个后缀预处理 $F(C)$ 等等。可以做到 $O(|S| \log |S| + 26|S|)$。

NOIP2020 字符串匹配

注意到 AB 是 S 的前缀，因此第一个想法是枚举 AB 的长度和 i，然后计算 AB 有多少个前缀 A 满足 $F(A) \leq F(C)$。

具体地，需要维护 cnt_k 表示当前有多少个前缀 A 满足 $F(A) = k$，利用哈希判定 $(AB)^i$ 是否是 S 的前缀，对每个后缀预处理 $F(C)$ 等等。可以做到 $O(|S| \log |S| + 26|S|)$。

接下来会发现不枚举 i 也是可以的。

NOIP2020 字符串匹配

注意到 AB 是 S 的前缀，因此第一个想法是枚举 AB 的长度和 i，然后计算 AB 有多少个前缀 A 满足 $F(A) \le F(C)$。

具体地，需要维护 cnt_k 表示当前有多少个前缀 A 满足 $F(A) = k$，利用哈希判定 $(AB)^i$ 是否是 S 的前缀，对每个后缀预处理 $F(C)$ 等等。可以做到 $O(|S| \log |S| + 26|S|)$。

接下来会发现不枚举 i 也是可以的。

$F(C)$ 的定义为出现奇数次的字符的数量，因此 $F(C)$ 只和 i 的奇偶性有关，当 i 为奇数时都和 $i = 1$ 相同，当 i 为偶数时 $F(C)$ 固定为 $F(S)$。

NOIP2020 字符串匹配

即使这样，也需要知道 i 最大能达到多少，用哈希不能做到 $O(1)$，但是 Z 函数就可以。

NOIP2020 字符串匹配

即使这样，也需要知道 i 最大能达到多少，用哈希不能做到 $O(1)$，但是 Z 函数就可以。

至此，我们考虑 $i=1$ 的情况，把所有前缀 A 的 $F(A)$ 插入树状数组，可以做到 $O(|S|\log 26)$。而 i 为偶数的情况更简单，容易做到 $O(|S|)$。

NOIP2020 字符串匹配

即使这样，也需要知道 i 最大能达到多少，用哈希不能做到 $O(1)$，但是 Z 函数就可以。

至此，我们考虑 $i=1$ 的情况，把所有前缀 A 的 $F(A)$ 插入树状数组，可以做到 $O(|S|\log 26)$。而 i 为偶数的情况更简单，容易做到 $O(|S|)$。

进一步优化需要用到一个事实，当 AB 的长度 $+1$ 时，$F(C)$ 只会 ± 1。把问题抽象出来就是：

- 维护一个序列 cnt，支持单点加和查询前缀和，每次查询的位置只会 ± 1。

NOIP2020 字符串匹配

即使这样，也需要知道 i 最大能达到多少，用哈希不能做到 $O(1)$，但是 Z 函数就可以。

至此，我们考虑 $i=1$ 的情况，把所有前缀 A 的 $F(A)$ 插入树状数组，可以做到 $O(|S|\log 26)$。而 i 为偶数的情况更简单，容易做到 $O(|S|)$。

进一步优化需要用到一个事实，当 AB 的长度 $+1$ 时，$F(C)$ 只会 ± 1。把问题抽象出来就是：

- 维护一个序列 cnt，支持单点加和查询前缀和，每次查询的位置只会 ± 1。

维护 $sum=\sum_{i=0}^{F(C)} cnt_i$，单点加和与 $F(C)\pm 1$ 时都能 $O(1)$ 修改 sum。最终做到了 $O(|S|)$。

CF961F k-substrings

给定一个长度为 n 的字符串 s，定义 k-子串为 $s[k, n-k+1]$。

对于字符串 s 的每个 k-子串，计算其最大奇数长度 border 的长度。

$n \le 10^6$，

提示：

题目链接

CF961F k-substrings

给定一个长度为 n 的字符串 s，定义 k-子串为 $s[k, n-k+1]$。

对于字符串 s 的每个 k-子串，计算其最大奇数长度 border 的长度。

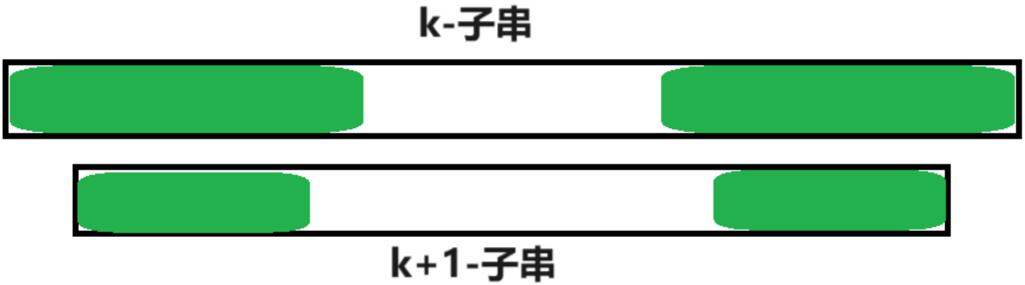
$n \le 10^6$，

提示：k-子串和 $k+1$-子串的答案有什么关系。

题目链接

CF961F k-substrings

观察 k-子串和 $k+1$-子串的答案，发现 $k+1$-子串可以继承 k-子串的 border。



CF961F k-substrings

观察 k-子串和 $k+1$-子串的答案，发现 $k+1$-子串可以继承 k-子串的 border。

由图可知 $ans_{k+1} \geq ans_k - 2$。

CF961F k-substrings

由图可知 $ans_{k+1} \geq ans_k - 2$。

即使这样，知道 ans_k 仍然不好求出 ans_{k+1}。

CF961F k-substrings

由图可知 $ans_{k+1} \geq ans_k - 2$。

即使这样，知道 ans_k 仍然不好求出 ans_{k+1}。

但是反过来就不一样了，由于 $ans_k \leq ans_{k+1} + 2$，可以从大到小枚举长度，用哈希 $O(1)$ 判定 border，从而 $O(ans_{k+1} - ans_k)$ 求出 ans_k。

CF961F k-substrings

由图可知 $ans_{k+1} \geq ans_k - 2$。

即使这样，知道 ans_k 仍然不好求出 ans_{k+1}。

但是反过来就不一样了，由于 $ans_k \leq ans_{k+1} + 2$，可以从大到小枚举长度，用哈希 $O(1)$ 判定 border，从而 $O(ans_{k+1} - ans_k)$ 求出 ans_k。

按照 $\lfloor \frac{n}{2} \rfloor$ 到 1 依次求出 ans_i，总复杂度 $O(n)$。

来源未知

给定字符串集合 S 表示禁用词集合，以及字符串 T，问将 T 划分为若干个非空子段（使得每个位置恰被一个子段包含）的方案数，满足划分出的每一段都不是禁用词。

答案对 10^9+7 取模。

$\sum_{s\in S}|s|\leq 2\cdot 10^5, |T|\leq 2\cdot 10^5$

提示：

来源未知

给定字符串集合 S 表示禁用词集合，以及字符串 T，问将 T 划分为若干个非空子段（使得每个位置恰被一个子段包含）的方案数，满足划分出的每一段都不是禁用词。

答案对 10^9+7 取模。

$\sum_{s\in S}|s| \le 2\cdot 10^5, |T| \le 2\cdot 10^5$

提示：对 S 建 AC 自动机。

来源未知

设 dp_i 表示将 $[1, i]$ 划分，并满足条件的方案数，那么

$$dp_i = \sum_{j=0}^{i-1} dp_j - \sum_{T[j,i] \in S} dp_{j-1}$$

来源未知

设 dp_i 表示将 $[1, i]$ 划分，并满足条件的方案数，那么

$$dp_i = \sum_{j=0}^{i-1} dp_j - \sum_{T[j,i]\in S} dp_{j-1}$$

注意 $T[j, i] \in S$ 这个条件，正是 AC 自动机可以处理的。对 S 建立 AC 自动机后，满足 $T[j, i] \in S$ 的所有 $T[j, i]$ 都可以表达为 fail 树上的一条链。

来源未知

设 dp_i 表示将 $[1,i]$ 划分，并满足条件的方案数，那么

$$dp_i=\sum_{j=0}^{i-1}dp_j-\sum_{T[j,i]\in S}dp_{j-1}$$

注意 $T[j,i]\in S$ 这个条件，正是 AC 自动机可以处理的。对 S 建立 AC 自动机后，满足 $T[j,i]\in S$ 的所有 $T[j,i]$ 都可以表达为 fail 树上的一条链。

另一方面，$\sum_{s\in S}|s|\le 2\cdot 10^5$ 说明了这些 $T[j,i]$ 的数量不能超过 $O(\sqrt{\sum|s|})$，进一步说明 fail 树的任意一条链上至多只有 $O(\sqrt{\sum|s|})$ 个结点代表了实际的字符串。

来源未知

设 dp_i 表示将 $[1,i]$ 划分，并满足条件的方案数，那么

$$dp_i = \sum_{j=0}^{i-1} dp_j - \sum_{T[j,i]\in S} dp_{j-1}$$

注意 $T[j,i] \in S$ 这个条件，正是 AC 自动机可以处理的。对 S 建立 AC 自动机后，满足 $T[j,i] \in S$ 的所有 $T[j,i]$ 都可以表达为 fail 树上的一条链。

另一方面，$\sum_{s\in S}|s| \le 2\cdot 10^5$ 说明了这些 $T[j,i]$ 的数量不能超过 $O(\sqrt{\sum|s|})$，进一步说明 fail 树的任意一条链上至多只有 $O(\sqrt{\sum|s|})$ 个结点代表了实际的字符串。

通过对 fail 树上的每个结点预处理上方的第一个“实际”结点，转移的时候就可以快速找出所有 $T[j,i] \in S$。

来源未知

设 dp_i 表示将 $[1, i]$ 划分，并满足条件的方案数，那么

$$dp_i = \sum_{j=0}^{i-1} dp_j - \sum_{T[j,i]\in S} dp_{j-1}$$

注意 $T[j, i] \in S$ 这个条件，正是 AC 自动机可以处理的。对 S 建立 AC 自动机后，满足 $T[j, i] \in S$ 的所有 $T[j, i]$ 都可以表达为 fail 树上的一条链。

另一方面，$\sum_{s\in S} |s| \le 2 \cdot 10^5$ 说明了这些 $T[j, i]$ 的数量不能超过 $O(\sqrt{\sum |s|})$，进一步说明 fail 树的任意一条链上至多只有 $O(\sqrt{\sum |s|})$ 个结点代表了实际的字符串。

通过对 fail 树上的每个结点预处理上方的第一个“实际”结点，转移的时候就可以快速找出所有 $T[j, i] \in S$。

复杂度 $O(n\sqrt{n})$。

CF914F Substrings in a String

给定一个字符串 s，处理 q 次查询，每次查询有以下两种形式之一：

- 1 i c —将字符串的第 i 个字符改为 c
- 2 l r y —考虑从位置 l 到 r 的子串，输出 y 在其中作为子串出现的次数

$|s| \leq 10^5$，$q \leq 10^5$，第二种查询的所有 y 长度之和不超过 10^5

提示：

题目链接

CF914F Substrings in a String

给定一个字符串 s，处理 q 次查询，每次查询有以下两种形式之一：

- 1 i c —将字符串的第 i 个字符改为 c
- 2 l r y —考虑从位置 l 到 r 的子串，输出 y 在其中作为子串出现的次数

$|s| \leq 10^5$，$q \leq 10^5$，第二种查询的所有 y 长度之和不超过 10^5

提示：复杂度 $O(\frac{n^2}{w})$。

题目链接

CF914F Substrings in a String

对于每次询问，可以使用 bitset 求出 y 在 s 中的出现位置集合（左端点）。

CF914F Substrings in a String

对于每次询问，可以使用 bitset 求出 y 在 s 中的出现位置集合（左端点）。

具体地，设 pos_c 表示字符 c 在 s 中的出现位置集合，那么 y 在 s 中的出现位置集合为：

$$\text{AND}_{i=1}^{|y|} pos_{y_i} >> (i-1)$$

该结果在 $[l, r-|y|+1]$ 中 1 的数量就是答案。

CF914F Substrings in a String

对于每次询问，可以使用 bitset 求出 y 在 s 中的出现位置集合（左端点）。

具体地，设 pos_c 表示字符 c 在 s 中的出现位置集合，那么 y 在 s 中的出现位置集合为：

$$\text{AND}_{i=1}^{|y|} pos_{y_i} >> (i-1)$$

该结果在 $[l, r-|y|+1]$ 中 1 的数量就是答案。

思考：怎么提取 bitset 在一个区间中 1 的数量。

CF914F Substrings in a String

对于每次询问，可以使用 bitset 求出 y 在 s 中的出现位置集合（左端点）。

具体地，设 pos_c 表示字符 c 在 s 中的出现位置集合，那么 y 在 s 中的出现位置集合为：

$$\text{AND}_{i=1}^{|y|} pos_{y_i} >> (i-1)$$

该结果在 $[l, r-|y|+1]$ 中 1 的数量就是答案。

思考：怎么提取 bitset 在一个区间中 1 的数量。

单次询问复杂度为 $O(|y| \cdot \frac{n}{w})$，总复杂度为 $O(\frac{n^2}{w})$。

HDU7084 Pty loves string

给定一个长度为 n 的字符串 s，需要回答 q 次询问。每次询问给出两个整数 x 和 y，表示取 s 的前缀 $s[1:x]$ 和后缀 $s[n-y+1:n]$ 拼接成一个新字符串 t。对于每次询问，计算字符串 t 在字符串 s 中出现的次数。

$n, q \leq 2 \times 10^5$

提示：

题目链接

HDU7084 Pty loves string

给定一个长度为 n 的字符串 s，需要回答 q 次询问。每次询问给出两个整数 x 和 y，表示取 s 的前缀 $s[1:x]$ 和后缀 $s[n-y+1:n]$ 拼接成一个新字符串 t。对于每次询问，计算字符串 t 在字符串 s 中出现的次数。

$n, q \leq 2 \times 10^5$

提示：会用到 KMP 的 fail 数组。

题目链接

HDU7084 Pty loves string

考虑一个出现位置 $s[l..r]$，会满足 $s[l..l+x-1]$ 和 $s[1..x]$ 相等，且 $s[r-y+1..r]$ 和 $s[n-y+1..n]$ 相等，且 $l+x=r-y+1$。

HDU7084 Pty loves string

考虑一个出现位置 $s[l..r]$，会满足 $s[l..l+x-1]$ 和 $s[1..x]$ 相等，且 $s[r-y+1..r]$ 和 $s[n-y+1..n]$ 相等，且 $l+x=r-y+1$。

将一个出现位置画出来后，就能发现很简单的判定条件：设 $i=l+x$，那么 x,y 分别是 $s[1..i-1]$ 和 $s[i..n]$ 的 border。

HDU7084 Pty loves string

考虑 KMP 做完从 i 向 $fail_i$ 连一条边，这样我们会得到一棵树，一个 $s[1,x]$ 在 i 处出现（右端点）当且仅当 i 属于 x 的子树。

HDU7084 Pty loves string

考虑 KMP 做完从 i 向 $fail_i$ 连一条边，这样我们会得到一棵树，一个 $s[1,x]$ 在 i 处出现（右端点）当且仅当 i 属于 x 的子树。

对反串做同样的事情，一个 $s[n-y+1,n]$ 在 i 处出现（左端点）当且仅当 i 属于 y 的子树。

HDU7084 Pty loves string

考虑 KMP 做完从 i 向 $fail_i$ 连一条边，这样我们会得到一棵树，一个 $s[1,x]$ 在 i 处出现（右端点）当且仅当 i 属于 x 的子树。

对反串做同样的事情，一个 $s[n-y+1,n]$ 在 i 处出现（左端点）当且仅当 i 属于 y 的子树。

那么问题可以变为：有两棵树，问两个子树内有多少个点编号相同?

HDU7084 Pty loves string

考虑 KMP 做完从 i 向 $fail_i$ 连一条边，这样我们会得到一棵树，一个 $s[1,x]$ 在 i 处出现（右端点）当且仅当 i 属于 x 的子树。

对反串做同样的事情，一个 $s[n-y+1,n]$ 在 i 处出现（左端点）当且仅当 i 属于 y 的子树。

那么问题可以变为：有两棵树，问两个子树内有多少个点编号相同？

这个问题可以转化为二维数点，用数据结构（扫描线 + 树状数组）维护即可。

复杂度 $O(n\log n)$。

谢谢大家！