字符串

租酥雨

2021 年 7 月 17 日

一些约定

- 如非特殊说明，所有字符串的下标均从 1 开始编号。
- σ 表示字符集大小，一般默认为 26。
- $S+T$ 表示字符串 S 与字符串 T 的拼接。
- $|S|$ 表示字符串 S 的长度。
- S_i 表示字符串 S 的第 i 个字符。
- $S_{i\ldots j}$ 表示字符串 S 第 i 个字符到第 j 个字符组成的子串。
- pre_i 表示字符串 S 长度为 i 的前缀，即 $S_{1\ldots i}$。
- suf_i 表示字符串 S 长度为 i 的后缀，即 $S_{|S|-i+1\ldots|S|}$。
- 定义一个串存在长度为 i 的 border 当且仅当 $pre_i = suf_i$。
- $lcp(i,j)$ 表示 suf_i 与 suf_j 的最长公共前缀。
- $lcs(i,j)$ 表示 pre_i 与 pre_j 的最长公共后缀。

outline

- 最小循环表示法
- manacher
- ex-kmp
- Hash
- kmp
- Trie
- Aho-Corasick automaton
- suffix array
- suffix automaton

最小循环表示法

给一个字符串 S，求一个 i 使 $S_{i\ldots|S|}+S_{1\ldots i-1}$ 的字典序最小，要求时间复杂度线性。

最小循环表示法

给一个字符串 S，求一个 i 使 $S_{i\ldots|S|}+S_{1\ldots i-1}$ 的字典序最小，要求时间复杂度线性。

将 S 倍长，维护两个指针 i,j 表示最小循环表示的起始位置，暴力求出 $lcp(i,j)$，设其为 k，则比较 S_{i+k} 和 S_{j+k}，不失一般性地假设 $S_{i+k}<S_{j+k}$，则 $[j,j+k]$ 范围内的所有下标均不可能成为最小循环表示的起始位置，直接令 $j\leftarrow j+k+1$ 后继续比较，直至 $[1,n]$ 内仅剩下一个起始位置。

由于指针 i,j 始终是递增的，因而该算法的时间复杂度为 $O(|S|)$。

最小循环表示法

以下的 s 已经被倍长，n 表示原串长。

```
int i = 1, j = 2;
while(i <= n && j <= n){
    int k = 0;
    while(k <= n && s[i + k] == s[j + k])
        ++k;
    if(s[i + k] < s[j + k])
        j += k + 1;
    else
        i += k + 1;
    j += (i == j);
}
```

manacher

manacher 算法可以对串 S 的每个位置 i，求出以其为中心的最长奇回文串长度。若需要求所有回文串，则可以在串 S 的任意两个相邻字符中间插入一种新的字符。

记 len_i 表示以位置 i 为中心的最长奇回文串半径，考虑从左往右依次求 len_i。记录前缀中 $i + len_i$ 的最大值 mx 及其对应的位置 pos，当需要计算 len_j 时，根据回文的对称性可以利用到之前已得到的信息，即

$$len_j \geq \min(len_{2pos-j}, mx - j)$$

这样每次暴力扩展都必然导致 mx 增大，从而使复杂度均摊 $O(|S|)$。

manacher

```
int mx = 0, pos = 0;
for(int i = 1; i <= n; ++i){
    len[i] = mx > i ? min(len[2 * pos - i], mx - i) : 1;
    while(s[i - len[i]] == s[i + len[i]])
        ++len[i];
    if(i + len[i] > mx)
        mx = i + len[i], pos = i;
}
```

ex-kmp

ex-kmp 又称 Z-algorithm，可以对串 S 求出所有 $lcp(1,i), i \in [2,|S|]$，即原串与所有后缀的最长公共前缀。

为什么先讲 ex-kmp 后讲 kmp？是因为 ex-kmp 跟 kmp 没什么太大关系，反倒是和 manacher 很相似。

类似 manacher，记 len_i 表示 $lcp(1,i)$，同时记录 $i+len_i$ 的最大值 mx 及其对应位置 pos，可以发现 $len_j \geq \min(len_{j-pos+1}, mx-j)$，其余部分与 manacher 几乎完全相同。

ex-kmp

```
int mx = 1, pos = 1;
for(int i = 2; i <= n; ++i){
    lcp[i] = mx > i ? min(lcp[i - pos + 1], mx - i) : 0;
    while(s[1 + lcp[i]] == s[i + lcp[i]])
        ++lcp[i];
    if(i + lcp[i] > mx)
        pos = i, mx = i + lcp[i];
}
```

hash

把一个字符串映射到一个数。通过构造 hash 函数以尽量减少冲突，理想情况是使映射趋于完全随机。

常见的构造函数形如：

$$hash(S) = \sum_{i=1}^{|S|} S_i \times Base^{|S|-i} \mod p$$

p 可以取一个大质数，也可以取 2^{32} 或 2^{64}，不过值得注意的是确定存在并且已经构造出了能使 $p = 2^{64}$ 产生 hash 冲突的字符串。

kmp

kmp 算法解决的是字符串匹配中单串匹配单串的问题。具体的，这个问题可以被描述为：给出一个长度为 n 的文本串 S 和一个长度为 m 的模式串 T，问 T 在 S 中出现了多少次，分别在哪些位置出现了。

朴素做法复杂度 $O(nm)$：枚举 S 的开头位置，$O(m)$ 遍历比较两串。

在一次比较失败后，我们放弃了以比较部分的全部信息，直接选取 S 的下一个位置继续匹配。这在一定程度上造成了浪费。

kmp

举个例子，假设文本串 $S = abbaabbbabaa$，模式串 $T = abbaaba$，从 S 的第一位开始匹配：

kmp

举个例子，假设文本串 $S = abbaabbbabaa$，模式串 $T = abbaaba$，从 S 的第一位开始匹配：

$abbaab\,bbabaa$
$abbaaba$

kmp

举个例子，假设文本串 $S = abbaabbbabaa$，模式串 $T = abbaaba$，从 S 的第一位开始匹配：

$abbaab\,bbabaa$
$abbaaba$

匹配了 6 位，在第 7 位上失配了。按照朴素做法，我们会直接放弃“已匹配 6 位”的信息，转而从 S 的第二位开始匹配。这其中浪费掉了很多已知的信息，造成了复杂度过高，由此，kmp 算法应运而生。

nxt 数组

kmp 的本质是对每一个 i 求出最大的 $j < i$ 满足 $S_{i-j+1\ldots i} = pre_j$（换言之即最长的相同前后缀），记之为 nxt_i。

举个例子，对于字符串 $aababaabaccc$ 来说，它的前缀 $aababaaba$ 的相同前后缀有 $aaba$、a 以及空串，所以它对应的 nxt 就等于 4。

nxt 数组的构造

在解决前面说过的字符串匹配问题时，需要先对 T 串构造 nxt 数组。nxt 数组的构造方式是增量构造，即在已经得到 $nxt_1 \ldots nxt_{i-1}$ 的情况下，求 nxt_i。

nxt 数组的构造

在解决前面说过的字符串匹配问题时，需要先对 T 串构造 nxt 数组。nxt 数组的构造方式是增量构造，即在已经得到 $nxt_1 ... nxt_{i-1}$ 的情况下，求 nxt_i。
如果 nxt_{i-1} 后一个字符与第 i 个字符相同，即 $T_{nxt_{i-1}+1} = T_i$，那么 $nxt_i = nxt_{i-1} + 1$。

nxt 数组的构造

在解决前面说过的字符串匹配问题时，需要先对 T 串构造 nxt 数组。nxt 数组的构造方式是增量构造，即在已经得到 $nxt_1 ... nxt_{i-1}$ 的情况下，求 nxt_i。
如果 nxt_{i-1} 后一个字符与第 i 个字符相同，即 $T_{nxt_{i-1}+1} = T_i$，那么 $nxt_i = nxt_{i-1} + 1$。
否则我们需要找到对于 $i-1$ 而言第二长的相同前后缀，这个实际上是 $nxt_{nxt_{i-1}}$，同理第三长的是 $nxt_{nxt_{nxt_{i-1}}}$，以此类推。

nxt 数组的构造

在解决前面说过的字符串匹配问题时，需要先对 T 串构造 nxt 数组。nxt 数组的构造方式是增量构造，即在已经得到 $nxt_1 ... nxt_{i-1}$ 的情况下，求 nxt_i。

如果 nxt_{i-1} 后一个字符与第 i 个字符相同，即 $T_{nxt_{i-1}+1} = T_i$，那么 $nxt_i = nxt_{i-1} + 1$。

否则我们需要找到对于 $i-1$ 而言第二长的相同前后缀，这个实际上是 $nxt_{nxt_{i-1}}$，同理第三长的是 $nxt_{nxt_{nxt_{i-1}}}$，以此类推。

我们按照长度顺序依次判断 $i-1$ 的某个相同前后缀的下一位是否与 T_i 相同，注意可能并不存在这样的前后缀，那么 nxt_i 将会等于 0。

nxt 数组的构造

在解决前面说过的字符串匹配问题时，需要先对 T 串构造 nxt 数组。nxt 数组的构造方式是增量构造，即在已经得到 $nxt_1 ... nxt_{i-1}$ 的情况下，求 nxt_i。
如果 nxt_{i-1} 后一个字符与第 i 个字符相同，即 $T_{nxt_{i-1}+1} = T_i$，那么 $nxt_i = nxt_{i-1} + 1$。
否则我们需要找到对于 $i-1$ 而言第二长的相同前后缀，这个实际上是 $nxt_{nxt_{i-1}}$，同理第三长的是 $nxt_{nxt_{nxt_{i-1}}}$，以此类推。
我们按照长度顺序依次判断 $i-1$ 的某个相同前后缀的下一位是否与 T_i 相同，注意可能并不存在这样的前后缀，那么 nxt_i 将会等于 0。代码差不多长这样：

```
for(int i = 2, j = 0; i <= m; ++i){
    while(j && T[j+1] != T[i])
        j = nxt[j];
    if(T[j+1] == T[i])
        ++j;
    nxt[i] = j;
}
```

代码中的 j 表示的是 nxt_i 指针的移动。

解决原问题

在使用 nxt 数组解决匹配问题前，先考虑一下我们可以怎样优化匹配的复杂度？还是用前面的例子：

$abbaabbbabaa$

$abbaaba$

解决原问题

在使用 nxt 数组解决匹配问题前，先考虑一下我们可以怎样优化匹配的复杂度？还是用前面的例子：

$abbaabbbabaa$

$abbaaba$

对于已经匹配的 6 位，我们考虑：S 的前 6 个起始位置中，哪些还可能模式串的匹配？可以断言只有第 5 位还有可能，这是因为串 $abbaab$ 的相同前后缀只有 ab 和空串。

解决原问题

在使用 nxt 数组解决匹配问题前，先考虑一下我们可以怎样优化匹配的复杂度？还是用前面的例子：

$abbaabbbabaa$

$abbaaba$

对于已经匹配的 6 位，我们考虑：S 的前 6 个起始位置中，哪些还可能模式串的匹配？可以断言只有第 5 位还有可能，这是因为串 $abbaab$ 的相同前后缀只有 ab 和空串。

代码和构造 nxt 数组差不多：

```
for(int i = 1, j = 0; i <= n; ++i){
    while(j && T[j+1] != S[i])
        j = nxt[j];
    if(T[j+1] == S[i])
        ++j;
    if(j == m){
        //match success
    }
}
```

kmp 的时间复杂度

前面说了朴素做法的时间复杂度是 $O(nm)$，那么 kmp 算法又如何呢?

kmp 的时间复杂度

前面说了朴素做法的时间复杂度是 $O(nm)$，那么 kmp 算法又如何呢?

```
for(int i = 2, j = 0; i <= m; ++i){
    while(j && T[j+1] != T[i])
        j = nxt[j];
    if(T[j+1] == T[i])
        ++j;
    nxt[i] = j;
}
```

考虑构建 nxt 数组的部分，注意观察 `j` 指针的变化，可以发现至多只有 m 次 `++j` 操作，而每次 `j = nxt[j]` 都一定会使 `j` 变小，因此 `for` 循环中的 `while` 语句在整个算法流程中只会被执行不超过 m 次。

kmp 的时间复杂度

前面说了朴素做法的时间复杂度是 $O(nm)$，那么 kmp 算法又如何呢?

```
for(int i = 2, j = 0; i <= m; ++i){
    while(j && T[j+1] != T[i])
        j = nxt[j];
    if(T[j+1] == T[i])
        ++j;
    nxt[i] = j;
}
```

考虑构建 nxt 数组的部分，注意观察 `j` 指针的变化，可以发现至多只有 m 次 `++j` 操作，而每次 `j = nxt[j]` 都一定会使 `j` 变小，因此 `for` 循环中的 `while` 语句在整个算法流程中只会被执行不超过 m 次。

因此构造 nxt 数组的时间复杂度为 $O(m)$，类似地可以分析出匹配时的时间复杂度为 $O(n)$，因此算法的总时间复杂度为 $O(n+m)$。

kmp 的时间复杂度

前面说了朴素做法的时间复杂度是 $O(nm)$，那么 kmp 算法又如何呢?

```
for(int i = 2, j = 0; i <= m; ++i){
    while(j && T[j+1] != T[i])
        j = nxt[j];
    if(T[j+1] == T[i])
        ++j;
    nxt[i] = j;
}
```

考虑构建 nxt 数组的部分，注意观察 `j` 指针的变化，可以发现至多只有 m 次 `++j` 操作，而每次 `j = nxt[j]` 都一定会使 `j` 变小，因此 `for` 循环中的 `while` 语句在整个算法流程中只会被执行不超过 m 次。

因此构造 nxt 数组的时间复杂度为 $O(m)$，类似地可以分析出匹配时的时间复杂度为 $O(n)$，因此算法的总时间复杂度为 $O(n+m)$。

顺带一提，由于我们以 $O(m)$ 的时间复杂度构造了长度为 m 的 nxt 数组，因此有时也称求一个 nxt_i 的**均摊 (amortized)** 复杂度为 $O(1)$。

kmp 自动机

在 kmp 求 nxt 的基础上，额外求出 $trans_{i,j}$ 表示从 i 位置开始往后匹配一个 j 字符后会转移到什么状态，可以实现真正严格 $O(1)$ 而非均摊 $O(1)$ 的转移。但需要注意预处理 $trans$ 的复杂度是 $O(m\sigma)$ 的。

Trie

又称字典树，结构是除根节点以外每个点上有一个字符（一种说法是每条边上有一条字符），每个节点所代表的字符串就是从根节点出发到该节点路径上所有字符顺次连接形成的串。

Trie

又称字典树，结构是除根节点以外每个点上有一个字符（一种说法是每条边上有一条字符），每个节点所代表的字符串就是从根节点出发到该节点路径上所有字符顺次连接形成的串。

两个串的最长公共前缀长度即为对应的两点在 Trie 树上 LCA 的深度。

Aho-Corasick automaton

本质是 Trie 上实现 kmp 自动机。

Aho-Corasick automaton

本质是 Trie 上实现 kmp 自动机。

按照深度由浅到深做。对每个节点求出 $fail_i$ 表示这个点对应的串在 Trie 树中存在的最长严格后缀，构造方式也与 kmp 类似，通过不断跳 $fail$ 以找到一条与当前字符相同的出边。

Aho-Corasick automaton

本质是 Trie 上实现 kmp 自动机。

按照深度由浅到深做。对每个节点求出 $fail_i$ 表示这个点对应的串在 Trie 树中存在的最长严格后缀，构造方式也与 kmp 类似，通过不断跳 $fail$ 以找到一条与当前字符相同的出边。

```
for(int i = 0; i < 26; ++i)
    if(tr[0][i])
        Q.push(tr[0][i]);
while(!Q.empty()){
    int u = Q.front();
    Q.pop();
    for(int i = 0; i < 26; ++i)
        if(tr[u][i])
            fail[tr[u][i]] = tr[fail[u]][i], Q.push(tr[u][i]);
        else
            tr[u][i] = tr[fail[u]][i];
}
```

Aho-Corasick automaton

一般的写法是直接建出自动机，在构建时只需要从其 $fail_i$ 转移而来即可。

Aho-Corasick automaton

一般的写法是直接建出自动机，在构建时只需要从其 $fail_i$ 转移而来即可。

同时，根据 $i \to fail_i$ 可以建出一个树形结构，这棵树被称为 $fail$ 树，它具有一些优美的性质，具体来说是 $fail$ 树上存在祖孙关系的两点对应的字符串具有后缀关系。

suffix array

求一个字符串任意两个后缀的 lcp。本质上实现了对所有后缀的字典序排序。

用 Height_i 表示后缀排序中排名为 i 的后缀与排名为 $i+1$ 的后缀的 lcp 长度，那么排名为 l 的后缀与排名为 r 的后缀的 lcp 长度就是 $\min\limits_{l\le i<r} \mathrm{Height}_i$，可以通过 RMQ 算法实现 $O(n\log n)-O(1)$ 求区间最小值。

求后缀数组一般使用倍增 + 基数排序，复杂度为 $O(n\log n)$。

suffix automaton & suffix tree

SAM 是一个能接受 S 的所有后缀（也可以说是所有子串）的有限状态自动机。

对于一个 S 的字串 s，记 $endpos(s)$ 表示 s 在 S 中所有出现位置的下标集合。把 $endpos$ 集合相同视作一种等价关系，那么 S 的所有字串会被分成 $O(n)$ 个等价类，同一个等价类中的所有串均存在后缀关系，且长度是连续的。

后缀树的形态其实是把 S 的所有后缀插到 Trie 中，再把所有链缩掉得到的结果。后缀树的每一个节点就对应着一种 $endpos$ 集合，因此可以认为后缀树的每个节点都表示着某一段“连续”的字符串集合。对于任意两个节点 u, v，若 v 是 u 的祖先，则 $endpos(u) \subset endpos(v)$；若两者不存在祖先关系，则 $endpos(u) \cap endpos(v) = \varnothing$。

后缀自动机的构建

一般用增量法来构造 SAM。

(见板书)

这种建 SAM 方法的线性时间复杂度证明可以参考OI-Wiki。

后缀自动机的构建

```
void extend(char c){
    int v = last, u = ++tot;
    last = u; len[u] = len[v] + 1;
    while (v && !tr[v][c]) tr[v][c] = u, v = fa[v];
    if (!v) fa[u] = 1;
    else{
        int x = tr[v][c];
        if (len[x] == len[v] + 1) fa[u] = x;
        else{
            int y = ++tot;
            memcpy(tr[y], tr[x], sizeof(tr[y]));
            fa[y] = fa[x]; fa[x] = fa[u] = y;
            len[y] = len[v] + 1;
            while (v && tr[v][c] == x) tr[v][c] = y, v = fa[v];
        }
    }
}
```

snoi2019 字符串

description

给一个长度为 n 的字符串 S，记 S'_i 表示 S 删去第 i 个字符后得到的字符串，求 $S'_1, S'_2, \cdots, S'_n$ 字典序排序的结果。

constriction

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

snoi2019 字符串

description

给一个长度为 n 的字符串 S，记 S'_i 表示 S 删去第 i 个字符后得到的字符串，求 $S'_1, S'_2, \cdots, S'_n$ 字典序排序的结果。

constriction

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

solution

考虑比较 S'_i 和 S'_j 的字典序大小，发现只需要用到 $lcp(\min(i,j), \min(i,j)+1)$，即相邻两个后缀的 lcp。这个可以从后往前 $O(n)$ 预处理出来。

thupc2018A 绿绿与串串

description

对于字符串 S，定义运算 $f(S)$ 为将 S 的前 $|S|-1$ 个字符倒序接在 S 后面形成的长为 $2|S|-1$ 的新字符串。
现给出字符串 T，询问有哪些串长度不超过 $|T|$ 的串 S 经过若干次 f 运算后得到的串包含 T 作为前缀。

constriction

$1 \leq |T| \leq 5 \times 10^6$.

thupc2018A 绿绿与串串

description

对于字符串 S，定义运算 $f(S)$ 为将 S 的前 $|S|-1$ 个字符倒序接在 S 后面形成的长为 $2|S|-1$ 的新字符串。
现给出字符串 T，询问有哪些串长度不超过 $|T|$ 的串 S 经过若干次 f 运算后得到的串包含 T 作为前缀。

constriction

$1 \leq |T| \leq 5 \times 10^6$.

solution

合法的串 S 一定是 T 的前缀。
先用 manacher 求出每个位置的回文半径，从后往前处理，一个位置合法当且仅当其回文半径右侧达到 $|T|$，或者左侧达到 1 且右侧位置合法。

ctsc2014 企鹅 QQ

description

给 n 个长度相同且互不相同的串，询问有多少对串仅在一个位置上不同。

constriction

$1 \leq n \leq 30000, 1 \leq |S_i| \leq 200.$

ctsc2014 企鹅 QQ

description

给 n 个长度相同且互不相同的串，询问有多少对串仅在一个位置上不同。

constriction

$1 \leq n \leq 30000, 1 \leq |S_i| \leq 200.$

solution

直接枚举哪一位不同，hash 前后缀即可。

cqoi2014 通配符匹配

description

给出一个带通配符的字符串 S，通配符分两种，一种可以匹配恰好一个字符，另一种可以匹配任意个（包括 0 个）字符。再给出 n 个串 T_i，询问 T_i 能否与 S 匹配。

constriction

$1 \leq n \leq 100, 1 \leq |S|, |T_i| \leq 10^5, 0 \leq$ 通配符个数 ≤ 10.

cqoi2014 通配符匹配

description

给出一个带通配符的字符串 S，通配符分两种，一种可以匹配恰好一个字符，另一种可以匹配任意个（包括 0 个）字符。再给出 n 个串 T_i，询问 T_i 能否与 S 匹配。

constriction

$1 \le n \le 100, 1 \le |S|, |T_i| \le 10^5, 0 \le$ 通配符个数 ≤ 10.

solution

设 $f_{i,j}$ 表示前 i 个通配符匹配 T 的前 j 个字符是否可行，用 hash 判断字符串相等即可。转移“匹配任意个字符”的通配符时需要做一次前缀和。

hnoi2008 GT 考试

description

给一个数字串 S，求有多少长度为 n 的数字串 T 不包含 S 作为子串。

constriction

$1 \leq |S| \leq 100, 1 \leq n \leq 10^9$.

hnoi2008 GT 考试

description

给一个数字串 S，求有多少长度为 n 的数字串 T 不包含 S 作为子串。

constriction

$1 \leq |S| \leq 100, 1 \leq n \leq 10^9$.

solution

对 S 建出 kmp 自动机，把匹配到串 S 的每个位置视作一个状态，转移是在串的末尾加一个字符，相当于是在 kmp 自动机上走一条出边，矩阵快速幂转移即可。

noi2011 阿狸的打字机

description

给你一棵 n 个节点的 Trie 树，q 次询问 Trie 树上 x 节点代表的字符串在 y 节点代表的字符串中出现了多少次。

constraint

$1 \leq n, q \leq 10^6$.

noi2011 阿狸的打字机

description

给你一棵 n 个节点的 Trie 树，q 次询问 Trie 树上 x 节点代表的字符串在 y 节点代表的字符串中出现了多少次。

constraint

$1 \leq n, q \leq 10^6$.

solution

问题等价于问 y 节点代表的字符串有多少个前缀包含 x 作为后缀，也就是问 Trie 树上根节点到 y 路径上的所有点中有多少点在 $fail$ 树上 x 的子树里，树状数组维护即可。

bzoj3413 匹配

description

给定一个模板串 S 和若干个匹配串 T，求 T 在 S 上暴力匹配所需要的比较次数（匹配成功一次即停止）。
暴力匹配指的是复杂度为 $O(|S| \times |T|)$ 的字符串匹配算法。

constraint

$|S|, |T| \leq 10^5, \sum |T| \leq 3 \times 10^6$

bzoj3413 匹配

description

给定一个模板串 S 和若干个匹配串 T，求 T 在 S 上暴力匹配所需要的比较次数（匹配成功一次即停止）。
暴力匹配指的是复杂度为 $O(|S| \times |T|)$ 的字符串匹配算法。

constraint

$|S|, |T| \leq 10^5, \sum |T| \leq 3 \times 10^6$

solution

首先求出 T 在 S 中第一次出现的位置。
然后问题就大致转化为了：求 T 与 S 在第一次匹配位置之前的所有后缀的 lcp 长度之和。

bzoj3413 匹配

description

给定一个模板串 S 和若干个匹配串 T，求 T 在 S 上暴力匹配所需要的比较次数（匹配成功一次即停止）。
暴力匹配指的是复杂度为 $O(|S| \times |T|)$ 的字符串匹配算法。

constraint

$|S|, |T| \leq 10^5, \sum |T| \leq 3 \times 10^6$

solution

首先求出 T 在 S 中第一次出现的位置。
然后问题就大致转化为了：求 T 与 S 在第一次匹配位置之前的所有后缀的 lcp 长度之和。
这个东西其实等价于：T 的每个前缀在第一次匹配位置之前的出现次数。
线段树合并维护 $endpos$ 集合即可。

谢谢大家！

祝大家学业有成！