# 题目

给定一个字符串 s，返回 s 的不同子字符串的个数。

字符串的 子字符串 是由原字符串删除开头若干个字符（可能是 0 个）并删除结尾若干个字符（可能是 0 个）形成的字符串。

示例 1：

输入：s = "aabbaba"

输出：21

解释：不同子字符串的集合是 ["a","b","aa","bb","ab","ba","aab","abb","bab","bba","aba","aabb","abba","bbab","baba","aabba","abbab","bbaba","aabbab","abbaba","aabbaba"]

示例 2：

输入：s = "abcdefg"

输出：28

提示：

1 <= s.length <= 500

s 由小写英文字母组成。

进阶：你可以以 O(n) 时间复杂度解决此问题吗？

# 分析

要解决“计算字符串中不同子字符串的个数”的问题，我们需要找出所有可能的连续子字符串，并统计其中不重复的数量。核心挑战是高效地存储和判断子字符串的唯一性，避免重复计数。

思路分析

字符串的子字符串数量为 `n\*(n+1)/2`（`n` 为字符串长度），因为长度为 `n` 的字符串有 `n` 个长度为1的子串、`n-1` 个长度为2的子串，以此类推，总和为 `n+(n-1)+...+1 = n\*(n+1)/2`。

我们需要从这些子串中筛选出不重复的部分，关键是\*\*高效去重\*\*。常见的方法有：

1、哈希集合存储子串：直接生成所有子串并加入哈希集合，利用集合的唯一性自动去重，最后返回集合大小。

2、后缀自动机：一种更高效的数据结构，能在 `O(n)` 时间内统计不同子串的数量（进阶方案）。

对于题目中 `n≤500` 的约束，哈希集合方法实现简单且效率足够，以下先介绍这种方法。

## 方法一：哈希集合存储子串（直观易懂）

思路

遍历所有可能的子串：通过双重循环，外层循环控制子串的起始索引 i，内层循环控制子串的结束索引 j（i≤j < n），提取子串 s[i..j]。

去重存储：将提取的子串加入哈希集合（自动去重）。

结果：哈希集合的大小即为不同子字符串的个数。

代码：

class Solution {

public:

int countDistinct(string s) {

int n = s.size();

unordered\_set<string> substrs; // 存储所有不重复的子串

// 遍历所有可能的子串

for (int i = 0; i < n; ++i) { // 子串起始索引

for (int j = i; j < n; ++j) { // 子串结束索引

substrs.insert(s.substr(i, j - i + 1)); // 提取子串并加入集合

}

}

return substrs.size(); // 集合大小即为不同子串的个数

}

};

复杂度分析：

时间复杂度：O(n³)。双重循环遍历子串（O(n²)），每次插入哈希集合需计算子串哈希值（O(n)），总时间为 O(n³)。对于 n=500，500³=1.25e8次操作，可接受。

空间复杂度：O(n³)。最坏情况下所有子串均不重复（如“abcdefg”），需存储n\*(n+1)/2个子串，每个子串最长为n，总空间为O(n³)。

## 方法二：后缀自动机（进阶，O(n) 时间复杂度）

对于进阶要求的 O(n) 时间复杂度，后缀自动机是最优选择。它通过压缩存储所有子串的公共部分，高效统计不同子串的数量。

后缀自动机原理

后缀自动机是一种能表示字符串所有后缀的最小有限状态机，其核心是通过“状态”和“转移”记录子串的出现情况。每个状态代表一组等价的子串（endpos 集合相同），不同状态对应不同的子串集合。

不同子串的总数 = 所有状态的 (长度上限 - 长度下限) 之和。

代码：

struct State {

int len; // 该状态表示的子串的最大长度

int link; // 后缀链接

unordered\_map<char, int> next; // 转移映射（字符→状态索引）

};

class Solution {

public:

int countDistinct(string s) {

int n = s.size();

if (n == 0) return 0;

vector<State> st;

st.reserve(2 \* n); // 后缀自动机的状态数最多为 2n-1

int last = 0;

st.push\_back({0, -1, {}}); // 初始状态

int total = 0; // 累计不同子串的数量

for (char c : s) {

// 创建新状态 curr，初始长度为 last 状态的长度 + 1

int curr = st.size();

st.push\_back({st[last].len + 1, 0, {}});

int p = last;

// 沿着后缀链接更新转移

while (p != -1 && !st[p].next.count(c)) {

st[p].next[c] = curr;

p = st[p].link;

}

if (p == -1) {

// p 到达初始状态的后缀链接（-1），curr 的后缀链接指向初始状态

st[curr].link = 0;

} else {

int q = st[p].next[c];

if (st[p].len + 1 == st[q].len) {

// q 状态的最大长度恰好是 p 的长度 + 1，curr 的后缀链接指向 q

st[curr].link = q;

} else {

// 分裂 q 状态为 clone

int clone = st.size();

st.push\_back({st[p].len + 1, st[q].link, st[q].next});

// 更新 p 及其后缀链接链上指向 q 的转移为 clone

while (p != -1 && st[p].next[c] == q) {

st[p].next[c] = clone;

p = st[p].link;

}

// 更新 q 和 curr 的后缀链接指向 clone

st[q].link = clone;

st[curr].link = clone;

}

}

// 累加当前状态贡献的新子串数量（curr.len - link.len）

total += st[curr].len - st[st[curr].link].len;

last = curr;

}

return total;

}

};

复杂度分析

时间复杂度：O(n)。每个字符处理时，后缀链接的遍历和状态分裂操作均为线性级，整体复杂度与字符串长度成正比。

空间复杂度：O(n)。状态数最多为2n-1，每个状态的转移映射存储的字符数有限（26个小写字母），总空间为线性级。

总结

对于n≤500的约束，方法一（哈希集合）实现简单，易于理解，完全能满足需求。

对于更大的 n（如 n=1e5），方法二（后缀自动机）是最优选择，能在O(n)时间内高效解决问题，符合进阶要求。

两种方法均能正确统计不同子字符串的个数，可根据实际场景选择。