检查字符串是否出现

> 给一个文本串 T 和多个模式串 P，我们要检查字符串 P 是否作为 T 的一个子串出现。

我们在 $O(|T|)$ 的时间内对文本串 T 构造后缀自动机。为了检查模式串 P 是否在 T 中出现，我们沿转移（边）从 t_0 开始根据 P 的字符进行转移。如果在某个点无法转移下去，则模式串 P 不是 T 的一个子串。如果我们能够这样处理完整个字符串 P，那么模式串在 T 中出现过。

对于每个字符串 P，算法的时间复杂度为 $O(|P|)$。此外，这个算法还找到了模式串 P 在文本串中出现的最大前缀长度。

不同子串个数

> 给一个字符串 S，计算不同子串的个数。

对字符串 S 构造后缀自动机。

每个 S 的子串都相当于自动机中的一些路径。因此不同子串的个数等于自动机中以 t_0 为起点的不同路径的条数。

考虑到 SAM 为有向无环图，不同路径的条数可以通过动态规划计算。即令 d_v 为从状态 v 开始的路径数量（包括长度为零的路径），则我们有如下递推方程：

$$d_v = 1 + \sum_{w:(v,w,c)\in DAWG} d_w$$

即，d_v 可以表示为所有 v 的转移的末端的和。

所以不同子串的个数为 $d_{t_0} - 1$（因为要去掉空子串）。

总时间复杂度为：$O(|S|)$。

另一种方法是利用上述后缀自动机的树形结构。每个节点对应的子串数量是 $\operatorname{len}(i) - \operatorname{len}(\operatorname{link}(i))$，对自动机所有节点求和即可。

所有不同子串的总长度

> 给定一个字符串 S，计算所有不同子串的总长度。

本题做法与上一题类似，只是现在我们需要考虑分两部分进行动态规划：不同子串的数量 d_v 和它们的总长度 ans_v。

我们已经在上一题中介绍了如何计算 d_v。ans_v 的值可以通过以下递推式计算：

$$ans_v = \sum_{w:(v,w,c)\in DAWG} d_w + ans_w$$

我们取每个邻接结点 w 的答案，并加上 d_w（因为从状态 v 出发的子串都增加了一个字符）。

算法的时间复杂度仍然是 $O(|S|)$。

同样可以利用上述后缀自动机的树形结构。每个节点对应的所有后缀长度是 $\frac{\operatorname{len}(i)\times(\operatorname{len}(i)+1)}{2}$，减去其 link 节点的对应值就是该节点的净贡献，对自动机所有节点求和即可。

字典序第 k 大子串

> 给定一个字符串 S。多组询问，每组询问给定一个数 K_i，查询 S 的所有子串中字典序第 K_i 大的子串。

解决这个问题的思路可以从解决前两个问题的思路发展而来。字典序第 k 大的子串对应于 SAM 中字典序第 k 大的路径，因此在计算每个状态的路径数后，我们可以很容易地从 SAM 的根开始找到第 k 大的路径。

预处理的时间复杂度为 $O(|S|)$，单次查询的复杂度为 $O(|ans| \cdot |\Sigma|)$（其中 ans 是查询的答案，$|\Sigma|$ 为字符集的大小）。

> 虽然该题是后缀自动机的经典题，但实际上这题由于涉及字典序，用后缀数组做最方便。

最小循环移位

> 给定一个字符串 S。找出字典序最小的循环移位。

容易发现字符串 $S + S$ 包含字符串 S 的所有循环移位作为子串。

所以问题简化为在 $S + S$ 对应的后缀自动机上寻找最小的长度为 $|S|$ 的路径，这可以通过平凡的方法做到：我们从初始状态开始，贪心地访问最小的字符即可。

总的时间复杂度为 $O(|S|)$。

出现次数

> 对于一个给定的文本串 T，有多组询问，每组询问给一个模式串 P，回答模式串 P 在字符串 T 中作为子串出现了多少次。

利用后缀自动机的树形结构，进行 dfs 即可预处理每个节点的终点集合大小。在自动机上查找模式串 P 对应的节点，如果存在，则答案就是该节点的终点集合大小；如果不存在，则答案为 0.

以下为原方法：

> 对文本串 T 构造后缀自动机。
>
> 接下来做预处理：对于自动机中的每个状态 v，预处理 cnt_v，使之等于 $\operatorname{endpos}(v)$ 集合的大小。事实上，对应同一状态 v 的所有子串在文本串 T 中的出现次数相同，这相当于集合 endpos 中的位置数。
>
> 然而我们不能明确的构造集合 endpos，因此我们只考虑它们的大小 cnt。
>
> 为了计算这些值，我们进行以下操作。对于每个状态，如果它不是通过复制创建的（且它不是初始状态 t_0），我们将它的 cnt 初始化为 1。然后我们按它们的长度 len 降序遍历所有状态，并将当前的 cnt_v 的值加到后缀链接指向的状态上，即：
>
> $$cnt_{\operatorname{link}(v)} += cnt_v$$
>
> 这样做每个状态的答案都是正确的。
>
> 为什么这是正确的？不是通过复制获得的状态，恰好有 $|T|$ 个，并且它们中的前 i 个在我们插入前 i 个字符时产生。因此对于每个这样的状态，我们在它被处理时计算它们所对应的位置的数量。因此我们初始将这些状态的 cnt 的值赋为 1，其它状态的 cnt 值赋为 0。
>
> 接下来我们对每一个 v 执行以下操作：$cnt_{\operatorname{link}(v)} += cnt_v$。其背后的含义是，如果有一个字符串 v 出现了 cnt_v 次，那么它的所有后缀也在完全相同的地方结束，即也出现了 cnt_v 次。

> 为什么我们在这个过程中不会重复计数（即把某些位置数了两次）呢？因为我们只将一个状态的位置添加到 **一个** 其它的状态上，所以一个状态不可能以两种不同的方式将其位置重复地指向另一个状态。
>
> 因此，我们可以在 $O(|T|)$ 的时间内计算出所有状态的 cnt 的值。
>
> 最后回答询问只需要查找值 cnt_t，其中 t 为模式串对应的状态，如果该模式串不存在答案就为 0。单次查询的时间复杂度为 $O(|P|)$。

第一次出现的位置

> 给定一个文本串 T，多组查询。每次查询字符串 P 在字符串 T 中第一次出现的位置（P 的开头位置）。

我们构造一个后缀自动机。我们对 SAM 中的所有状态预处理位置 firstpos。即，对每个状态 v 我们想要找到第一次出现这个状态的末端的位置 $\text{firstpos}[v]$。换句话说，我们希望先找到每个集合 endpos 中的最小的元素（显然我们不能显式地维护所有 endpos 集合）。

为了维护 firstpos 这些位置，我们将原函数扩展为 `sam_extend()`。当我们创建新状态 cur 时，我们令：

$$\text{firstpos}(cur) = \text{len}(cur) - 1$$

；当我们将结点 q 复制到 $clone$ 时，我们令：

$$\text{firstpos}(clone) = \text{firstpos}(q)$$

（因为值的唯一的其它选项 $\text{firstpos}(cur)$ 显然太大了）。

那么查询的答案就是 $\text{firstpos}(t) - |P| + 1$，其中 t 为对应字符串 P 的状态。单次查询只需要 $O(|P|)$ 的时间。

所有出现的位置

> 问题同上，这一次需要查询文本串 T 中模式串出现的所有位置。

利用后缀自动机的树形结构，遍历子树，一旦发现终点节点就输出。

以下为原解法：

> 我们还是对文本串 T 构造后缀自动机。与上一个问题相似，我们为所有状态计算位置 firstpos。
>
> 如果 t 为对应于模式串 T 的状态，显然 $\text{firstpos}(t)$ 为答案的一部分。需要查找的其它位置怎么办？我们使用了含有字符串 P 的自动机，我们还需要将哪些状态纳入自动机呢？所有对应于以 P 为后缀的字符串的状态。换句话说我们要找到所有可以通过后缀链接到达状态 t 的状态。
>
> 因此为了解决这个问题，我们需要为每一个状态保存一个指向它的后缀引用列表。查询的答案就包含了对于每个我们能从状态 t 只使用后缀引用进行 DFS 或 BFS 的所有状态的 firstpos 值。
>
> 这种变通方案的时间复杂度为 $O(answer(P))$，因为我们不会重复访问一个状态（因为对于仅有一个后缀链接指向一个状态，所以不存在两条不同的路径指向同一状态）。
>
> 我们只需要考虑两个可能有相同 endpos 值的不同状态。如果一个状态是由另一个复制而来的，则这种情况会发生。然而，这并不会对复杂度分析造成影响，因为每个状态至多被复制一次。
>
> 此外，如果我们不从被复制的节点输出位置，我们也可以去除重复的位置。事实上对于一个状态，如果经过被复制状态可以到达，则经过原状态也可以到达。因此，如果我们给每个状态记录标记 `is_clone` 来代表这个状态是不是被复制出来的，我们就可以简单地忽略掉被复制的状态，只输出其它所有状态的 $firstpos$ 的值。

> 以下是大致的实现：

```cpp
struct state {
bool is_clone;
int first_pos;
std::vector<int> inv_link;
// some other variables
};

// 在构造 SAM 后
for (int v = 1; v < sz; v++) st[st[v].link].inv_link.push_back(v);

// 输出所有出现位置
void output_all_occurrences(int v, int P_length) {
if (!st[v].is_clone) cout << st[v].first_pos - P_length + 1 << endl;
for (int u : st[v].inv_link) output_all_occurrences(u, P_length);
}
```

最短的没有出现的字符串

> 给定一个字符串 S 和一个特定的字符集，我们要找一个长度最短的没有在 S 中出现过的字符串。

我们在字符串 S 的后缀自动机上做动态规划。

令 d_v 为节点 v 的答案，即，我们已经处理完了子串的一部分，当前在状态 v，想找到不连续的转移需要添加的最小字符数量。计算 d_v 非常简单。如果不存在使用字符集中至少一个字符的转移，则 $d_v = 1$。否则添加一个字符是不够的，我们需要求出所有转移中的最小值：

$$d_v = 1 + \min_{w:(v,w,c)\in SAM} d_w$$

问题的答案就是 d_{t_0}，字符串可以通过计算过的数组 d 逆推回去。

两个字符串的最长公共子串

> 给定两个字符串 S 和 T，求出最长公共子串，公共子串定义为在 S 和 T 中都作为子串出现过的字符串 X。

我们对字符串 S 构造后缀自动机。

我们现在处理字符串 T，对于每一个前缀，都在 S 中寻找这个前缀的最长后缀。换句话说，对于每个字符串 T 中的位置，我们想要找到这个位置结束的 S 和 T 的最长公共子串的长度。显然问题的答案就是所有 l 的最大值。

为了达到这一目的，我们使用两个变量，**当前状态** v 和 **当前长度** l。这两个变量描述当前匹配的部分：它的长度和它们对应的状态。

一开始 $v = t_0$ 且 $l = 0$，即，匹配为空串。

现在我们来描述如何添加一个字符 T_i 并为其重新计算答案：

- 如果存在一个从 v 到字符 T_i 的转移，我们只需要转移并让 l 自增一。
- 如果不存在这样的转移，我们需要缩短当前匹配的部分，这意味着我们需要按照后缀链接进行转移：

$$v = \text{link}(v)$$

与此同时，需要缩短当前长度。显然我们需要将 l 赋值为 $\text{len}(v)$，因为经过这个后缀链接后我们到达的状态所对应的最长字符串是一个子串。

- 如果仍然没有使用这一字符的转移，我们继续重复经过后缀链接并减小 l，直到我们找到一个转移或到达虚拟状态 -1（这意味着字符 T_i 根本没有在 S 中出现过，所以我们设置 $v = l = 0$）。

这一部分的时间复杂度为 $O(|T|)$，因为每次移动我们要么可以使 l 增加一，要么可以在后缀链接间移动几次，每次都减小 l 的值。

代码实现：

```
string lcs(const string &S, const string &T) {
  sam_init();
  for (int i = 0; i < S.size(); i++) sam_extend(S[i]);

  int v = 0, l = 0, best = 0, bestpos = 0;
  for (int i = 0; i < T.size(); i++) {
    while (v && !st[v].next.count(T[i])) {
      v = st[v].link;
      l = st[v].length;
    }
    if (st[v].next.count(T[i])) {
      v = st[v].next[T[i]];
      l++;
    }
    if (l > best) {
      best = l;
      bestpos = i;
    }
  }
  return T.substr(bestpos - best + 1, best);
}
```

多个字符串间的最长公共子串

> 给定 k 个字符串 S_i。我们需要找到它们的最长公共子串，即作为子串出现在每个字符串中的字符串 X。

我们将所有的子串连接成一个较长的字符串 T，以特殊字符 D_i 分开每个字符串（一个字符对应一个字符串）：

$$T = S_1 + D_1 + S_2 + D_2 + \cdots + S_k + D_k.$$

然后对字符串 T 构造后缀自动机。

现在我们需要在自动机中找到存在于所有字符串 S_i 中的一个字符串，这可以通过使用添加的特殊字符完成。注意如果 S_j 包含了一个子串，则 SAM 中存在一条从包含字符 D_j 的子串而不包含以其它字符 $D_1, \ldots, D_{j-1}, D_{j+1}, \ldots, D_k$ 开始的路径。

因此我们需要计算可达性，即对于自动机中的每个状态和每个字符 D_i，是否存在这样的一条路径。这可以容易地通过 DFS 或 BFS 及动态规划计算。之后，问题的答案就是状态 v 的字符串 $\text{longest}(v)$ 中存在所有特殊字符的路径。