西安电子科技大学 XIDIAN UNIVERSITY | 程序设计竞赛实训基地 Programming Contest Training Base

字符串
String

Leachim

计算机科学与技术学院

July 13, 2021

第 I 部分

前言

前队伍：rand
本科 2019 级计算机科学与技术专业
ICPC 银 +CCPC 银
Codeforces ID:

内容简介
Note

- AC 自动机
- 后缀数组 (SA)
- 后缀自动机 (SAM)
- 广义后缀自动机 (GSAM)

课前说明
Anouncement

由于本人很菜，如果在讲的时候有任何错误或者遗漏，恳请各位大佬指出。也欢迎大家随时就课件内容与我交流。
[P.S.] 该课件为公开课件，欢迎白嫖。

字符串算法属于那种入门门槛很低，而且都很套路。毕竟~~大家肯定都懂~~如何：

字符串算法属于那种入门门槛很低，而且都很套路。毕竟~~大家肯定都懂~~如何：

- AC 自动机 fail 树上 dfs 序建可持久化线段树
- 后缀自动机 next 指针 DAG 图上求 SG 函数

字符串算法属于那种入门门槛很低，而且都很套路。毕竟~~大家肯定都懂~~如何：

- AC 自动机 fail 树上 dfs 序建可持久化线段树
- 后缀自动机 next 指针 DAG 图上求 SG 函数

总的来说学习起来性价比还是很高的。

字符串算法属于那种入门门槛很低，而且都很套路。毕竟~~大家肯定都懂~~如何：

- AC 自动机 fail 树上 dfs 序建可持久化线段树
- 后缀自动机 next 指针 DAG 图上求 SG 函数

总的来说学习起来性价比还是很高的。
对吧？

本节课代码模板：

- AC 自动机：
  https://github.com/Michaelwmx/Cpp_Template/blob/master/AC-Automaton.cpp
- 倍增实现后缀数组 SA：
  https://github.com/Michaelwmx/Cpp_Template/blob/master/Suffix_Array.cpp
- 后缀自动机 SAM：
  https://github.com/Michaelwmx/Cpp_Template/blob/master/Suffix-Automaton.cpp
- 广义后缀自动机 GSAM：
  https://github.com/Michaelwmx/Cpp_Template/blob/master/General_Suffix-Automaton.cpp

第 II 部分

AC 自动机 (AC-Automaton)

大家肯定都学过 KMP 了，KMP 解决的是单个模式串 s 去匹配某个匹配串 t，问出现了多少次。

大家肯定都学过 KMP 了，KMP 解决的是单个模式串 s 去匹配某个匹配串 t，问出现了多少次。
那么如果有多个模式串 $s_1, s_2, \ldots, s_l$，问在 t 中出现了多少次?

引入 Lead In

大家肯定都学过 KMP 了，KMP 解决的是单个模式串 s 去匹配某个匹配串 t，问出现了多少次。

那么如果有多个模式串 $s_1, s_2, \ldots, s_l$，问在 t 中出现了多少次?

如果利用 KMP 给每个模式串 s_i 建立 next 数组然后跑匹配，复杂度是 $O(l \times |t| + \sum_i |s_i|)$ 的，不能接受。

大家肯定都学过 KMP 了，KMP 解决的是单个模式串 s 去匹配某个匹配串 t，问出现了多少次。

那么如果有多个模式串 $s_1, s_2, \ldots, s_l$，问在 t 中出现了多少次？

如果利用 KMP 给每个模式串 s_i 建立 next 数组然后跑匹配，复杂度是 $O(l \times |t| + \sum_i |s_i|)$ 的，不能接受。

不妨我们试着不分开建 next 数组，看看能不能合并？比如考虑 Trie 树？

字典树和失配指针
Trie Tree & Fail Pointer

我们把所有的模式串 s_i 都合并成一个 Trie 树 S。

字典树和失配指针
Trie Tree & Fail Pointer

我们把所有的模式串 s_i 都合并成一个 Trie 树 S。
从根到每一个节点都代表一个前缀，与 KMP 类似的，我们需要找到其对应的失配指针（fail 指针）。

字典树和失配指针
Trie Tree & Fail Pointer

我们把所有的模式串 s_i 都合并成一个 Trie 树 S。
从根到每一个节点都代表一个前缀，与 KMP 类似的，我们需要找到其对应的失配指针（fail 指针）。
与 KMP 不同的是，我们现在要找的这个指针指向的是针对所有前缀中最长的可以匹配当前后缀的节点。

使用 BFS 遍历 Trie 树

使用 BFS 遍历 Trie 树
然后对于每一个节点的所有儿子，都和 KMP 一样不停的跳 fail 指针检查一下是否可以接上。

使用 BFS 遍历 Trie 树
然后对于每一个节点的所有儿子，都和 KMP 一样不停的跳 fail 指针检查一下是否可以接上。
然后放入队列里，就这样。

使用 BFS 遍历 Trie 树
然后对于每一个节点的所有儿子，都和 KMP 一样不停的跳 fail 指针检查一下是否可以接上。
然后放入队列里，就这样。
flukehn 说有个口诀：“fail 的儿子是儿子的 fail”

【模板】AC 自动机（简单版）

题目来源：【模板】AC 自动机（简单版）

题意：给定 n 个模式串 s_i 和一个文本串 t，求有多少个不同编号的模式串在文本串里出现过。

数据范围：$1 \le n \le 10^6, 1 \le |t| \le 10^6, 1 \le \sum_{i=1}^{n} |s_i| \le 10^6$

西安电子科技大学 XIDIAN UNIVERSITY | 程序设计竞赛实训基地 Programming Contest Training Base

第 III 部分

后缀数组 (Suffix Array)

定义 Defination

- 所谓后缀数组或 SA 数组，其实就是一个储存着后缀排序的数组。

定义 Defination

- 所谓后缀数组或 SA 数组，其实就是一个储存着后缀排序的数组。

举个例子: "abacd" 的所有后缀经过排序之后就是 {abacd,acd,bacd,cd,d}
我们把对应的下标写出来就变成了 SA[]={1,3,2,4,5}

定义 Defination

- 所谓后缀数组或 SA 数组，其实就是一个储存着后缀排序的数组。

举个例子: "abacd" 的所有后缀经过排序之后就是 {abacd,acd,bacd,cd,d}
我们把对应的下标写出来就变成了 SA[]={1,3,2,4,5}

值得一提的是，老生常谈的“后缀数组”往往直接和求后缀数组的算法挂钩。不过更加严格的来说，算法名字应该叫后缀排序 (Suffix Sort) 而不是后缀数组 (Suffix Array)。

定义 Defination

- 所谓后缀数组或 SA 数组，其实就是一个储存着后缀排序的数组。

举个例子: "abacd" 的所有后缀经过排序之后就是 {abacd,acd,bacd,cd,d}
我们把对应的下标写出来就变成了 SA[]={1,3,2,4,5}

值得一提的是，老生常谈的“后缀数组”往往直接和求后缀数组的算法挂钩。不过更加严格的来说，算法名字应该叫后缀排序 (Suffix Sort) 而不是后缀数组 (Suffix Array)。

不过也不奇怪，因为后缀数组最难的点在于怎么使用，而不是如何排序，有点像网络流。~~(不过 SA 的使用并不是今天重点)~~

1

11

后缀排序的算法有很多，以下是几种比较常见的：

后缀排序的算法有很多，以下是几种比较常见的：

1. 快速排序 + 二分 hash 求 LCP(最长公共前缀) 实现 cmp 函数。$O(n \log^2 n)$（易懂，没啥用，大家知道就好了）

后缀排序的算法有很多，以下是几种比较常见的：

1. 快速排序 + 二分 hash 求 LCP(最长公共前缀) 实现 cmp 函数。$O(n \log^2 n)$（易懂，没啥用，大家知道就好了）
2. 基数排序 + 倍增 $O(n \log n)$。（思想很好，经典算法，推荐学习）

1

11

后缀排序的算法有很多，以下是几种比较常见的：

1. 快速排序 + 二分 hash 求 LCP(最长公共前缀) 实现 cmp 函数。$O(n \log^2 n)$（易懂，没啥用，大家知道就好了）
2. 基数排序 + 倍增 $O(n \log n)$。（思想很好，经典算法，推荐学习）
3. 通过分治排序的 DC3 算法 $O(n)$。（常数略大，不推荐使用）

后缀排序的算法有很多，以下是几种比较常见的：

1. 快速排序 + 二分 hash 求 LCP(最长公共前缀) 实现 cmp 函数。$O(n \log^2 n)$（易懂，没啥用，大家知道就好了）
2. 基数排序 + 倍增 $O(n \log n)$。（思想很好，经典算法，推荐学习）
3. 通过分治排序的 DC3 算法 $O(n)$。（常数略大，不推荐使用）
4. 诱导排序的 SA-IS 算法 $O(n)$。（应该是最快的了，模板不长）

后缀排序的算法有很多，以下是几种比较常见的：

1. 快速排序 + 二分 hash 求 LCP(最长公共前缀) 实现 cmp 函数。$O(n \log^2 n)$（易懂，没啥用，大家知道就好了）
2. 基数排序 + 倍增 $O(n \log n)$。（思想很好，经典算法，推荐学习）
3. 通过分治排序的 DC3 算法 $O(n)$。（常数略大，不推荐使用）
4. 诱导排序的 SA-IS 算法 $O(n)$。（应该是最快的了，模板不长）
5. 通过线性构造后缀树求 SA，$O(n)$ 或 $O(n \log |\sum|)$。（这是唯一一个与字符集大小有关的 SA “线性”算法，一定注意。不推荐使用来单纯求 SA。但是推荐学习如何构造后缀树）

基数排序 + 倍增
Radix Sort+Doubling

基数排序 + 倍增的思想其实不难，只是代码写起来比较麻烦。

基数排序 + 倍增
Radix Sort+Doubling

基数排序 + 倍增的思想其实不难，只是代码写起来比较麻烦。大家一看下图肯定都懂了。

基数排序 + 倍增
Radix Sort+Doubling

基数排序 + 倍增的思想其实不难，只是代码写起来比较麻烦。大家一看下图肯定都懂了。

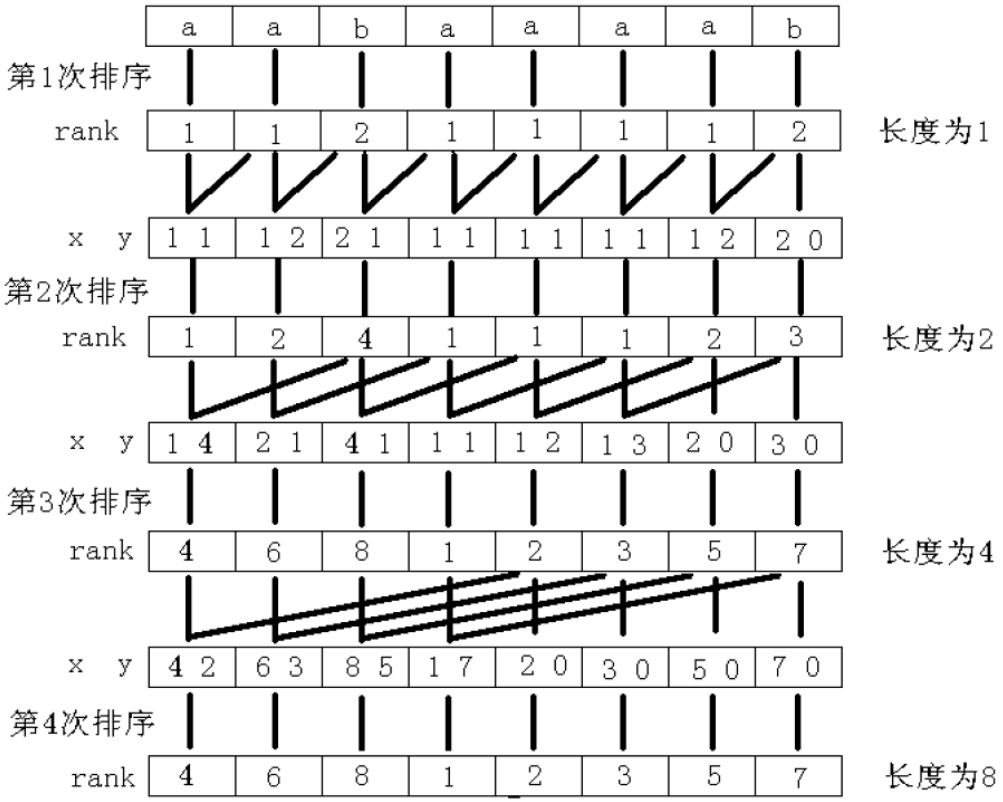

图 1: 后缀排序 (图源 oi-wiki)

基数排序 + 倍增
Radix Sort+Doubling

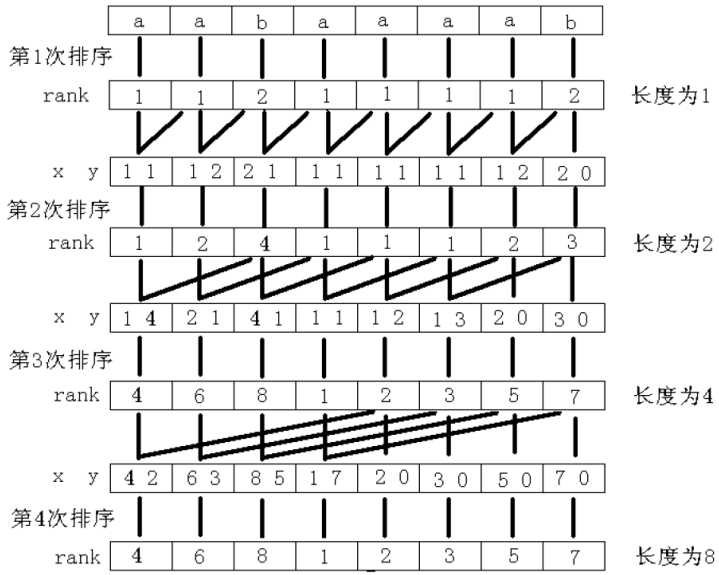

图 2: 后缀排序 (图源 oi-wiki)

就是说我们先通过桶排序把从 i 开头长度为 1 的字符串排序给写出来

然后就可以利用两个长度为 1 的拼成长度为 2 的，相当于是二级排序，使用桶就可以 $O(n)$ 处理出来。

以此类推每次都可以把长度扩充 2 倍，所以只需要 log 次就可以完成，总复杂度是 $O(n \log n)$。

最长公共子串
[SPOJ]1811

题目来源：LCS - Longest Common Substring

题意：给定 S, T 两个字符串，求 S 与 T 的最长公共子串。
其中 $|S|, |T| \leq 250000$

最长公共子串 [SPOJ]1811

引入后缀数组的一个经典应用，求 Height 数组。

最长公共子串
[SPOJ]1811

引入后缀数组的一个经典应用，求 Height 数组。

Height[] 数组的定义就是 $height[k] = LCP(S_{sa[k]}.., S_{sa[k-1]}..)$。

最长公共子串
[SPOJ]1811

引入后缀数组的一个经典应用，求 Height 数组。

Height[] 数组的定义就是 $height[k] = LCP(S_{sa[k]}.., S_{sa[k-1]}..)$。

用人话说就是，对于排序好的后缀来说，当前第 k 名与第 $k-1$ 名的 LCP。

最长公共子串
[SPOJ]1811

引入后缀数组的一个经典应用，求 Height 数组。

Height[] 数组的定义就是 $height[k] = LCP(S_{sa[k]}.., S_{sa[k-1]}..)$。

用人话说就是，对于排序好的后缀来说，当前第 k 名与第 $k-1$ 名的 LCP。

显然，利用 $height[rk[i]] - 1 \leq height[rk[i+1]]$ 的性质，我们可以在 $O(n)$ 的时间复杂度里面构造出来 Height 数组。

最长公共子串
[SPOJ]1811

引入后缀数组的一个经典应用，求 Height 数组。

Height[] 数组的定义就是 $height[k] = LCP(S_{sa[k]}.., S_{sa[k-1]}..)$。

用人话说就是，对于排序好的后缀来说，当前第 k 名与第 $k-1$ 名的 LCP。

显然，利用 $height[rk[i]] - 1 \leq height[rk[i+1]]$ 的性质，我们可以在 $O(n)$ 的时间复杂度里面构造出来 Height 数组。

当我们知道了一个字符串的 Height 数组之后，我们就可以在 $O(n)$ 的时间内，求出该串任意两个后缀的 LCP。

最长公共子串 [SPOJ]1811

引入后缀数组的一个经典应用，求 Height 数组。

Height[] 数组的定义就是 $height[k] = LCP(S_{sa[k]}.., S_{sa[k-1]}..)$。

用人话说就是，对于排序好的后缀来说，当前第 k 名与第 $k-1$ 名的 LCP。

显然，利用 $height[rk[i]] - 1 \leq height[rk[i+1]]$ 的性质，我们可以在 $O(n)$ 的时间复杂度里面构造出来 Height 数组。

当我们知道了一个字符串的 Height 数组之后，我们就可以在 $O(n)$ 的时间内，求出该串任意两个后缀的 LCP。

字符串任意两个后缀排序之间的所有后缀的 LCP 取并集，也就是这两个后缀之间的 LCP。很好理解，因为字符串比较就是从按位比较的。
也就是说，对于任意两个后缀，利用 Height 求 LCP 是 $O(n)$ 的

最长公共子串 [SPOJ]1811

回到这题上，我们知道一个字符串的任意两个后缀求 LCP 在有 Height 数组的情况下是 $O(n)$，那么两个字符串怎么办?

最长公共子串
[SPOJ]1811

回到这题上，我们知道一个字符串的任意两个后缀求 LCP 在有 Height 数组的情况下是 $O(n)$，那么两个字符串怎么办?

一个常见的套路就是对于整个 $S +' \#' + T$ 来求 SA，然后就变成一个串中两个不同后缀之间最大的 LCP 了。

最长公共子串 [SPOJ]1811

回到这题上，我们知道一个字符串的任意两个后缀求 LCP 在有 Height 数组的情况下是 $O(n)$，那么两个字符串怎么办?

一个常见的套路就是对于整个 $S +' \#' + T$ 来求 SA，然后就变成一个串中两个不同后缀之间最大的 LCP 了。

这个放置 $'\#'$ 在中间的套路在字符串题目中经常出现，可以做个笔记。

Infinite Fraction Path
[HDU]6223

题目来源：2017 ICPC 沈阳 G. Infinite Fraction Path

题目大意：给定一个字符集为 $'0'-'9'$ 字符串 S，连接 $S[i]$ 与 $S[(i^2+1)\%|S|]$。从任意一个 $S[i]$ 出发，经过之前连接的唯一的路径，可以走出一个无限长的字符串。
求走出的字符串中最大的字符串的前 $|S|$ 位

$1 \le N \le 150000$

Infinite Fraction Path
[HDU]6223

这题目原题的连边的比较特殊 $i->(i^2+1)\%|S|$，所以其实是可以暴力 + 剪枝过的。事实上当时打练习赛的时候，IGVA就是这么过的这题 orz。

Infinite Fraction Path
[HDU]6223

这题目原题的连边的比较特殊 $i->(i^2+1)\%|S|$，所以其实是可以暴力 + 剪枝过的。事实上当时打练习赛的时候，IGVA就是这么过的这题 orz。

不过如果我们将边改为随意连接，那么就不可以暴力了。

Infinite Fraction Path
[HDU]6223

这题目原题的连边的比较特殊 $i->(i^2+1)\%|S|$，所以其实是可以暴力 + 剪枝过的。事实上当时打练习赛的时候，IGVA就是这么过的这题 orz。

不过如果我们将边改为随意连接，那么就不可以暴力了。

我们可以借助后缀数组倍增求法的思想，依次算出以每一个点为起点长度为 $1, 2, 4, 8, ..., 2^k$ 拉出来字符串的排序，在最多求 $\lfloor \log n \rfloor + 1$ 次的情况下就可以算出最终排序。最终复杂度是 $O(n \log n)$

奇偶序列
[XDOJ]1522

题目来源：西安电子科技大学 2020 新生赛现场赛 J. 奇偶序列

题目大意：对于一个 01 串 $x[1...n]$。定义其奇偶序列 $y[1...n]$，$y[k] = (j-k) \times (k-i)$，其中 i 是最大的 $i < k$ 满足 $x[i] = x[k]$，j 是最小的 $j > k$ 满足 $x[j] \neq x[k]$，如果 i 或 j 不存在则 $y[k] = 0$。给定 01 串 s 求每个后缀串对应的奇偶序列的字典序排序。
$|s| \leq 10^6$

比如给定 “100101” 的话，后缀从后往前依次构造出来的序列为
$\{0\}, \{0,0\}, \{0,0,0\}, \{0,0,2,0\}, \{0,1,0,2,0\}, \{0,0,1,3,2,0\}$
最终排序结果就是 $\{6,5,4,1,3,2\}$。

奇偶序列
[XDOJ]1522

首先我们要会构造这个奇偶序列，题目暗示的很明显了从后往前构造，我们只需要维护当前最靠前的 0 和 1 的位置，每向前扫一个数，我们就对应的修改对应的 0 1 位置上的值就好。

奇偶序列
[XDOJ]1522

首先我们要会构造这个奇偶序列，题目暗示的很明显了从后往前构造，我们只需要维护当前最靠前的 0 和 1 的位置，每向前扫一个数，我们就对应的修改对应的 0 1 位置上的值就好。

那么在构造的时候就会发现，虽然说我们要排序若干个串并不是同一个串的后缀，但是相比较于最长那个串的对应长度的后缀，只有两个位置不同即当前第一次出现的 0 和 1 的位置，而且不同的这个位置的值是 0。

奇偶序列
[XDOJ]1522

首先我们要会构造这个奇偶序列，题目暗示的很明显了从后往前构造，我们只需要维护当前最靠前的 0 和 1 的位置，每向前扫一个数，我们就对应的修改对应的 0 1 位置上的值就好。

那么在构造的时候就会发现，虽然说我们要排序若干个串并不是同一个串的后缀，但是相比较于最长那个串的对应长度的后缀，只有两个位置不同即当前第一次出现的 0 和 1 的位置，而且不同的这个位置的值是 0。

接下来我们考虑这两个位置之间的值，很明显可以想到两个第一次出现的 $0,1$ 之间一定全都是 0 或 1，最后构造出来的奇偶序列前缀一定是 $0,4,3,2,1,0$ 这种形式，也就是这个前缀越长，排名越后。

奇偶序列
[XDOJ]1522

首先我们要会构造这个奇偶序列，题目暗示的很明显了从后往前构造，我们只需要维护当前最靠前的 0 和 1 的位置，每向前扫一个数，我们就对应的修改对应的 0 1 位置上的值就好。

那么在构造的时候就会发现，虽然说我们要排序若干个串并不是同一个串的后缀，但是相比较于最长那个串的对应长度的后缀，只有两个位置不同即当前第一次出现的 0 和 1 的位置，而且不同的这个位置的值是 0。

接下来我们考虑这两个位置之间的值，很明显可以想到两个第一次出现的 $0,1$ 之间一定全都是 0 或 1，最后构造出来的奇偶序列前缀一定是 $0,4,3,2,1,0$ 这种形式，也就是这个前缀越长，排名越后。

那么我们现在就可以把奇偶序列分成两段，一段是由 0 开头由 0 结尾的前缀，一段是与最长的奇偶序列共享的后缀，然后双关键字排序即可。

B-Suffix Array
[NowCoder]5666A

题目来源：2020 牛客多校第一场 A. B-Suffix Array

上一题是这题的改编版本，我就不讲这题了，有兴趣的可以看看课件我就跳过了。

B-Suffix Array
[NowCoder]5666A

题目来源：2020 牛客多校第一场 A. B-Suffix Array

题目大意：给定一个字符集为 $'a' - 'b'$ 的字符串 t。对于每一个 t 的后缀 $(t[i]..t[n])$，我们求一个等长序列 $(b_i[i]..b_i[n])$，使得 $b_i[j]$ 代表 $(t[i]..t[j-1])$ 中与 $t[j]$ 相同字符的最短距离 (若不存在即为 0)。
求每个后缀对应的序列的字典序排序。$\sum |t| \leq 10^6$

比如给定 $'abaab'$ 的话，后缀从后往前依次构造出来的序列为 $(0), (0,0), (0,1,0), (0,0,1,3), (0,0,2,1,3)$
最终排序结果就是 54213。

B-Suffix Array
[NowCoder]5666A

std 的做法是一个结论，是出题人叉姐根据别人论文中的一个结论出的题。这题靠着个结论可以转换成一道裸的 SA。但这很不赛场~~其实只是我不会~~。所以我这里介绍一种比较能在赛场上想出来的算法。

B-Suffix Array
[NowCoder]5666A

std 的做法是一个结论，是出题人叉姐根据别人论文中的一个结论出的题。这题靠着个结论可以转换成一道裸的 SA。但这很不赛场~~其实只是我不会~~。所以我这里介绍一种比较能在赛场上想出来的算法。

其实多自己构造几组数据就会发现，因为字符集大小为 2，所以最后构造出来的序列最多只有两个 0，而且剩余部分与原串对应的序列是一致的。而且，构造出来的序列的两个 0 之间一定全是 1。

B-Suffix Array
[NowCoder]5666A

std 的做法是一个结论，是出题人叉姐根据别人论文中的一个结论出的题。这题靠着个结论可以转换成一道裸的 SA。但这很不赛场~~其实只是我不会~~。所以我这里介绍一种比较能在赛场上想出来的算法。

其实多自己构造几组数据就会发现，因为字符集大小为 2，所以最后构造出来的序列最多只有两个 0，而且剩余部分与原串对应的序列是一致的。而且，构造出来的序列的两个 0 之间一定全是 1。

所以，所求的序列串的排序，相当于靠两个关键字排序。第一关键字为 01..10，第二关键字为第二个 0 后面的序列串的后缀。那问题就很简单了，对原串对应序列串求 SA 当作第二关键字，而第一关键字看 1 的长度排序即可。

B-Suffix Array
[NowCoder]5666A

值得一提的是，我这个算法在赛场上 TLE 了，后来换了别人的 SA 板子才过的。

B-Suffix Array
[NowCoder]5666A

值得一提的是，我这个算法在赛场上 TLE 了，后来换了别人的 SA 板子才过的。

事后查明发现是我自己的 SA 的板子里面有个非常愚蠢的操作，在 swap 的时候不是 swap 的指针，而是 swap 的整个数组，导致我在多组数据的时候复杂度是假的。

B-Suffix Array
[NowCoder]5666A

值得一提的是，我这个算法在赛场上 TLE 了，后来换了别人的 SA 板子才过的。

事后查明发现是我自己的 SA 的板子里面有个非常愚蠢的操作，在 swap 的时候不是 swap 的指针，而是 swap 的整个数组，导致我在多组数据的时候复杂度是假的。

所以大家一定要注意网上的板子，里面很有可能参杂了这种错误！大家小心。

如果有熟悉 SA 的同学可能会发现，我之前选的题并没有花大笔墨在 SA 的 Height 数组上面。事实上，能把 SA 玩出花的关键其实就是 Height 数组。不过，我个人并不喜欢用 SA 乱搞，相反，接下来要讲的 SAM 在这些乱搞中反而能展现更大的威力，所以我个人更喜欢用 SAM 乱搞。

如果有熟悉 SA 的同学可能会发现，我之前选的题并没有花大笔墨在 SA 的 Height 数组上面。事实上，能把 SA 玩出花的关键其实就是 Height 数组。不过，我个人并不喜欢用 SA 乱搞，相反，接下来要讲的 SAM 在这些乱搞中反而能展现更大的威力，所以我个人更喜欢用 SAM 乱搞。

不过有兴趣的同学可以在拓展阅读资料中获取更多的 SA 乱搞指南。

拓展阅读资料 Extension

- 后缀数组——处理字符串的有力工具 by 罗穗骞
- 后缀数组 by 许智磊

拓展阅读资料 Extension

- 后缀数组——处理字符串的有力工具 by 罗穗骞
- 后缀数组 by 许智磊

当然还有一些网络资料大家可以自行查找，不过从以上论文的年代就可以看出，SA 其实在算法竞赛圈子里面已经算是非常年长的算法了，并不能很好地适应如今的环境。（不过似乎 SA-IS 算法在算法竞赛圈里面似乎不算特别老?? 我没有仔细考究，如果有人感兴趣可以去搜搜）

第 IV 部分

后缀自动机 (Suffix Automaton)

后缀排序的出现让我们可以得以对一个字符串中的字串进行一些骚操作，但是有的时候这还不够。我们希望向 AC 自动机一样可以访问一个字符串中所有的子串咋办。

后缀排序的出现让我们可以得以对一个字符串中的字串进行一些骚操作，但是有的时候这还不够。我们希望向 AC 自动机一样可以访问一个字符串中所有的子串咋办。

如果对着字符串所有的后缀建立 AC 自动机的话复杂度肯定爆炸了，我们需要一个更好的数据结构来优化时空复杂度，所以就有了后缀自动机（SAM）。

定义与性质
Definations and Properties

后缀自动机（SAM）是一个由单个字符串构造的而成的 DAG，从根节点开始走到任意一条路径都是原串某一个字串，某一个字串也唯一对应了一条从根开始走的路径。

定义与性质
Definations and Properties

1

后缀自动机（SAM）是一个由单个字符串构造的而成的 DAG，从根节点开始走到任意一条路径都是原串某一个字串，某一个字串也唯一对应了一条从根开始走的路径。

28

这里可以理解成所有后缀所构造的 Trie 树，然后我们将一些可以合并的节点合并了。所以 Trie 树的节点唯一对应了一条路径，但是 SAM 的一个节点对应了多个路径，也就相应的对应了多个子串。

1

后缀自动机（SAM）是一个由单个字符串构造的而成的 DAG，从根节点开始走到任意一条路径都是原串某一个字串，某一个字串也唯一对应了一条从根开始走的路径。

28

这里可以理解成所有后缀所构造的 Trie 树，然后我们将一些可以合并的节点合并了。所以 Trie 树的节点唯一对应了一条路径，但是 SAM 的一个节点对应了多个路径，也就相应的对应了多个子串。

我们看一张图

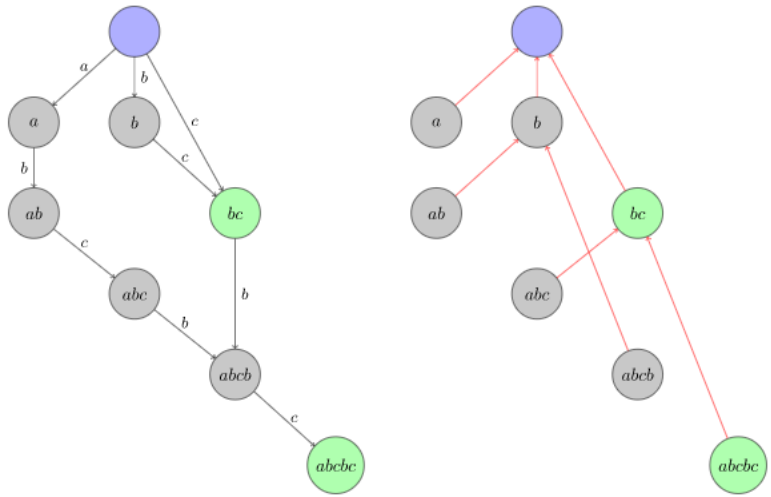

图 3: 串'abcbc' 的 SAM(图源 oi-wiki)

定义与性质
Definations and Properties

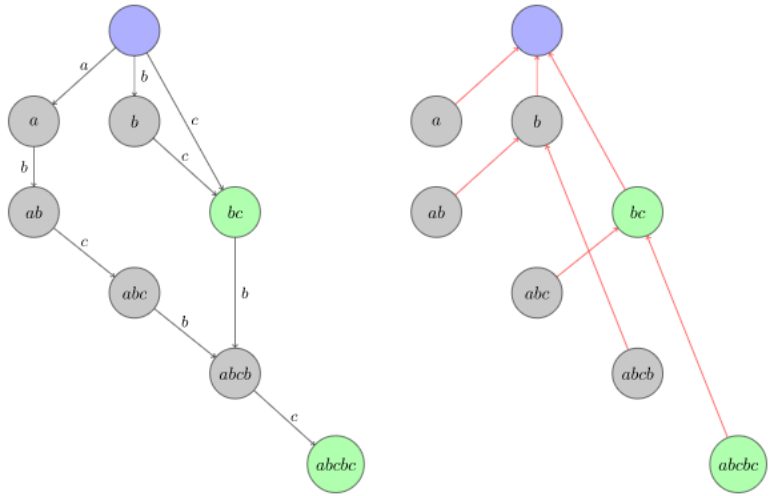

图 4: 串'abcbc' 的 SAM(图源 oi-wiki)

我们称每个节点为 endpos，他所代表的路径（子串）集合为 endpos 集合。什么情况下我们会把一系列子串放入到同一个集合里呢？当且仅当他们所有终止的位置都是完全一致的。

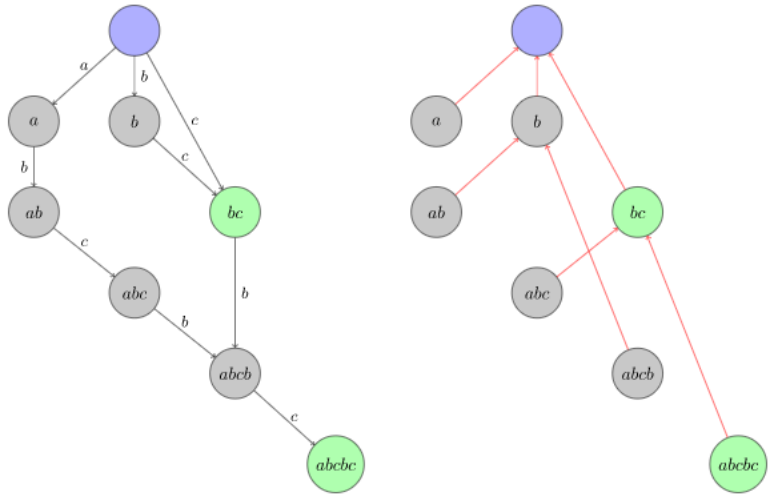

图 4: 串'abcbc' 的 SAM(图源 oi-wiki)

我们称每个节点为 endpos，他所代表的路径（子串）集合为 endpos 集合。什么情况下我们会把一系列子串放入到同一个集合里呢？当且仅当他们所有终止的位置都是完全一致的。

上图中对于一个多次出现的子串 bc 来说，bc 和 c 就应该处在同一个 endpos 集合，因为他们的终止点完全一致，所以他们被同一个节点所表示。

定义与性质
Definations and Properties

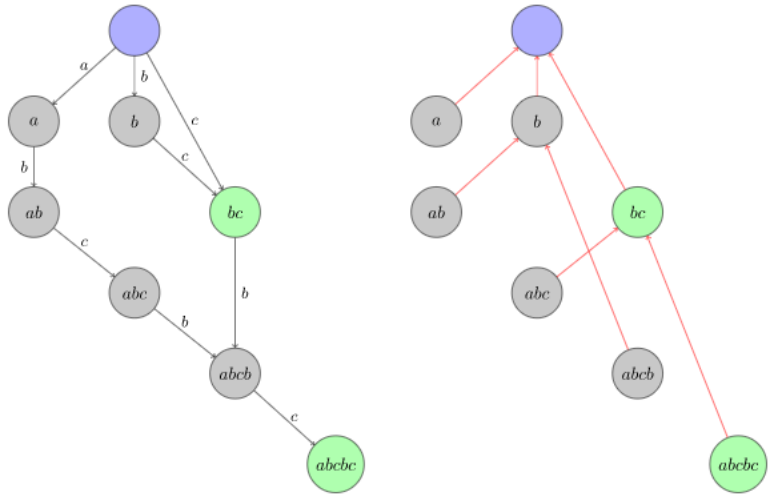

图 5: 串'abcbc' 的 SAM(图源 oi-wiki)

每个节点会有一个后缀链接 (link 指针) 指向 endpos 集合的子串的（最短的子串-1）的节点。比如某节点表示 {abcb,bcb,cb}，那么它的后缀链接指向表示 {b} 的节点。而 {b} 的节点指向根节点。

定义与性质
Definations and Properties

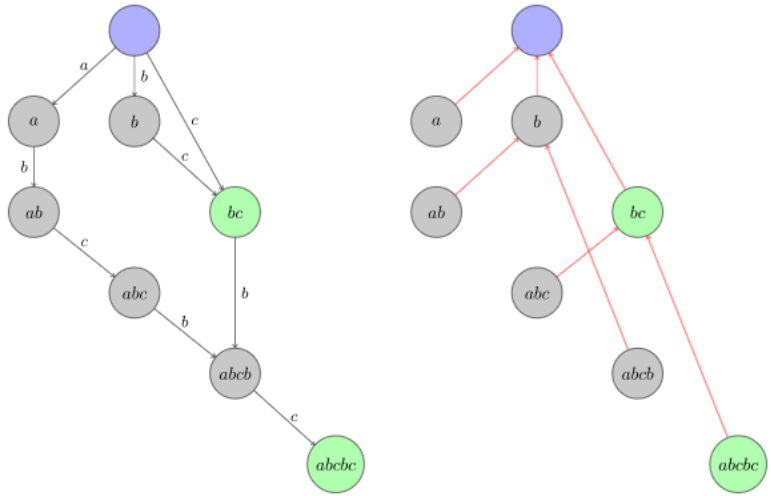

图 5: 串'abcbc' 的 SAM(图源 oi-wiki)

每个节点会有一个后缀链接 (link 指针) 指向 endpos 集合的子串的（最短的子串-1）的节点。比如某节点表示 {abcb,bcb,cb}，那么它的后缀链接指向表示 {b} 的节点。而 {b} 的节点指向根节点。

我们定义每个节点表示的子串中最长的长度为 len，比如 {abcb,bcb,cb} 的 $len = 4$。根据以上定义，我们可以看出。表示的子串中最长的长度为 $(len[link] + 1)$，所以表示的子串本质不同个数就是 $len - len[link]$。

定义与性质
Definations and Properties

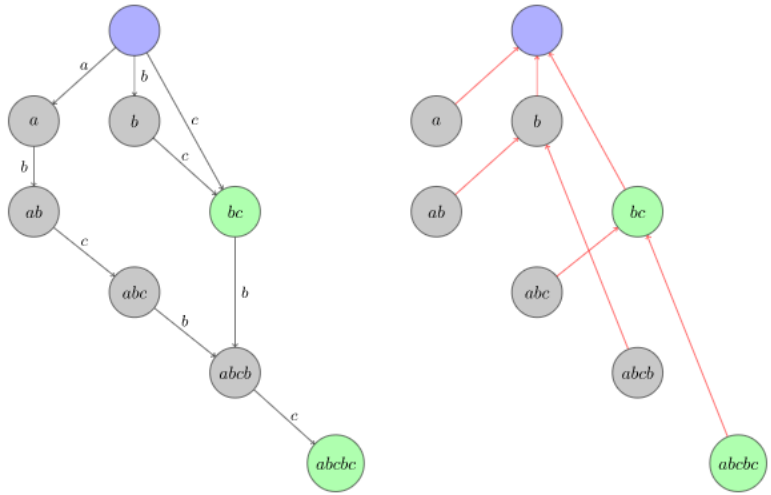

图 6: 串'abcbc' 的 SAM(图源 oi-wiki)

左图是后缀自动机的连边情况，是一个 DAG，通常用作匹配/DP。右图是 link 指针的情况，是一棵树（俗称 parent 树），通常用作失配/DP。

代码虽然不长，解释起来有点费劲。
大概就是要维护一个 last 节点，然后每次 insert 的时候要从 last 那里衍生出来
然后从 last 递归跳 link 指针，如果跳到的字符串没有对应新加入的儿子，就加上，然后继续跳
直到跳完或者遇到一个已经存在该儿子的节点，这个时候我们需要判断该节点与该节点的儿子是否表示的是相同的一些终止点，如果是的话我们可以直接把后缀链接连过去，如果不是的话我们需要把这个节点的儿子分裂成两个节点，然后把后缀链接指向分裂出来更短的那个节点，然后递归更新之前找到节点的 link 的儿子。

代码虽然不长，解释起来有点费劲。
大概就是要维护一个 last 节点，然后每次 insert 的时候要从 last 那里衍生出来
然后从 last 递归跳 link 指针，如果跳到的字符串没有对应新加入的儿子，就加上，然后继续跳
直到跳完或者遇到一个已经存在该儿子的节点，这个时候我们需要判断该节点与该节点的儿子是否表示的是相同的一些终止点，如果是的话我们可以直接把后缀链接连过去，如果不是的话我们需要把这个节点的儿子分裂成两个节点，然后把后缀链接指向分裂出来更短的那个节点，然后递归更新之前找到节点的 link 的儿子。

就不证明这个算法是正确的了，感兴趣的可以去自己查阅资料

复杂度介绍 Complexity

虽然我们常说后缀自动机是线性的，但是我还是要强调一遍，这个线性是与字符集大小挂钩的。

1. 在最简单的写法中 (每个节点对儿子开大小 $|\sum|$ 的数组)，此时时空复杂度都是 $O(n|\sum|)$ 的。
2. 如果说题目时间卡得很紧，可以考虑在开了儿子的数组的基础上，开一个链表。这样时间复杂度可以压到 $O(n)$，而空间复杂度还是 $O(n|\sum|)$ 的。(并不是很实用的 trick)。
3. 如果字符集很大或不是常数的时候，我们就需要在 SAM 节点里面用 map 等平衡树来储存孩子。这个时候时间复杂度就变成 $O(n\log|\sum|)$，而空间复杂度变成 $O(n)$。

虽然我们常说后缀自动机是线性的，但是我还是要强调一遍，这个线性是与字符集大小挂钩的。

1. 在最简单的写法中 (每个节点对儿子开大小 $|\sum|$ 的数组)，此时时空复杂度都是 $O(n|\sum|)$ 的。
2. 如果说题目时间卡得很紧，可以考虑在开了儿子的数组的基础上，开一个链表。这样时间复杂度可以压到 $O(n)$，而空间复杂度还是 $O(n|\sum|)$ 的。(并不是很实用的 trick)。
3. 如果字符集很大或不是常数的时候，我们就需要在 SAM 节点里面用 map 等平衡树来储存孩子。这个时候时间复杂度就变成 $O(n\log|\sum|)$，而空间复杂度变成 $O(n)$。

由于 SAM 其实常数并不是很大，字符集大小为常数的时候，一般来说可以轻易过 10^6 的题，所以下面不会考虑使用第 2 种的 trick 进行复杂度分析。

- [SDOI2016] 生成魔咒 (在线添加，询问本质不同子串个数)
- [Luogu]3804 【模板】后缀自动机 (求字符串中 len*cnt 最大的子串)
- [SPOJ]1811 LCS（求两个字符串的最长公共子串）
- [SPOJ]7258 SUBLEX（求本质不同排名第 k 小子串）
- [TJOI2015] 弦论（求第 k 小子串）

DNA
[Luogu]3763

题目来源：[TJOI2017] DNA

题目大意：给定字符集为 {'A','T','G','C'} 的两个字符串 S_0, S。求 S_0 中有多少个与 S 等长的子串，与 S 相差不超过 3 个位置。

$|S| \leq |S_0| \leq 10^5, 0 < T \leq 10$

DNA
[Luogu]3763

这是个模糊字符串匹配，普通的 KMP 是无法处理。

这是个模糊字符串匹配，普通的 KMP 是无法处理。

我们考虑对匹配串 S_0 建立 SAM，然后用模式串 S 来按位匹配 (从根 dfs 整个 SAM 的 DAG)。让 dfs(cur,cnt,len) 代表上一个节点为 cur，目前已经修改 cnt 次，当前匹配长度为 len。（也可以用类似 dp 一样 bfs）

DNA
[Luogu]3763

这是个模糊字符串匹配，普通的 KMP 是无法处理。

我们考虑对匹配串 S_0 建立 SAM，然后用模式串 S 来按位匹配 (从根 dfs 整个 SAM 的 DAG)。让 dfs(cur,cnt,len) 代表上一个节点为 cur，目前已经修改 cnt 次，当前匹配长度为 len。（也可以用类似 dp 一样 bfs）

虽然这个做法实际运行速度很快，但是这个算法的复杂度似乎并不好算？不是很会...就留给大家当作课后习题子。

回文串
[Luogu]3649

题目来源：[APIO2014] 回文串

题目大意：给定一个小写字母组成的字符串 S，求所有回文子串中，(出现次数 $\times$ 子串长度) 的最大值

$|S| \leq 3.0 \times 10^5$

回文串
[Luogu]3649

这明显是~~回文树~~回文串与 SAM 结合的题。

回文串
[Luogu]3649

这明显是~~回文树~~回文串与 SAM 结合的题。

SAM 构造出来的节点有个与回文串有关的性质，就是：每个节点所表示的点内最多只有一个回文串，而且该回文串一定是以当前节点为终点的最长的回文串，且是该回文串第一次出现。

回文串
[Luogu]3649

这明显是~~回文树~~回文串与 SAM 结合的题。

SAM 构造出来的节点有个与回文串有关的性质，就是：每个节点所表示的点内最多只有一个回文串，而且该回文串一定是以当前节点为终点的最长的回文串，且是该回文串第一次出现。

可以用反证法证明

1. 如果每个节点表示了多个回文串，那么短的回文串一定在长的回文串中出现了至少两个 endpos，与 endpos 的性质相悖。
2. 同样如果有多个回文子串结束于某个节点，那么次短的那个的 endpos 一定在之前节点也有。
3. 如此一来，每个回文子串第一次出现的终点，也只可能被该点所表示，就必定是以该点为终点的最长子串。

回文串
[Luogu]3649

所以这题直接用 manacher 求出来以每个点作为终点的最长回文子串的长度，然后再建立 SAM，每次插入之后检查以当前节点为结尾的回文串的最大长度是否在 last 节点的表示范围内，如果是的话就记录下。

回文串
[Luogu]3649

所以这题直接用 manacher 求出来以每个点作为终点的最长回文子串的长度，然后再建立 SAM，每次插入之后检查以当前节点为结尾的回文串的最大长度是否在 last 节点的表示范围内，如果是的话就记录下。
这里要记得在分裂节点的时候要判断分裂的节点到底是否包含了他之前记录的回文串。

回文串
[Luogu]3649

所以这题直接用 manacher 求出来以每个点作为终点的最长回文子串的长度，然后再建立 SAM，每次插入之后检查以当前节点为结尾的回文串的最大长度是否在 last 节点的表示范围内，如果是的话就记录下。
这里要记得在分裂节点的时候要判断分裂的节点到底是否包含了他之前记录的回文串。
最后扫描一遍 SAM 把表示的回文串的长度 $\times$ 出现次数，求个最大值即可。

SAM 其实是个潜能很大的东西，如果想进一步挖掘的话空间是很大的，尤其是结合一些数据结构可以做到一些很骚的在线操作，这里就不细讲了。

SAM 其实是个潜能很大的东西，如果想进一步挖掘的话空间是很大的，尤其是结合一些数据结构可以做到一些很骚的在线操作，这里就不细讲了。

同样，我这里虽然没有涉及到多串的题目，但是其实朴素 SAM 是可以处理多串题目的（接下来要讲的广义 SAM 的模板题，有一些是完全可以靠朴素 SAM 做，就是思维复杂度和代码量都会相对较高，不如直接建广义 SAM. 当然这并不是说就不需要大家学习，其中对 SAM 的运用还是很值得探究的）

拓展阅读资料 Extension

- 后缀自动机——陈立杰
- 后缀自动机 SAM——oi-wiki
- 后缀自动机及其应用——张天扬

第 V 部分

广义后缀自动机 (General Suffix Automaton)

定义 Defination

由于之前 SAM 只是对单串建立，有时候不能满足邪恶的出题人。于是就有了广义 SAM。广义 SAM 其实就是对着多串的 Trie 树建 SAM，性质与 SAM 类似。SAM 和广义 SAM 之间的性质上的区别和 KMP 和 AC 自动机之间类似。

不过大家在学习的时候不要被这个名字所吓到，如果从实现的角度来讲，GSAM 当成 SAM 的一个使用小 trick 都不为过。所以学了 SAM，只需要多花几分钟学一下 GSAM 不为过。

浅谈实现

目前网络上流传的广义 SAM 的写法主要有 3 种（来自于网络资料，没有严格查证）

1. 将多个字符串用'#' 等符号隔开，然后变成单长串，建立单串 SAM。
   (非常直接，很好理解，不推荐写。)
2. 多个串按顺序 extend，当且仅当在 extend 一个新串的时候，将 last 指针设置到 root。
   (非常好写，有细节需要注意，推荐学习。)
3. 对多个串建立 Trie 树，对 Trie 树进行 bfs 的同时改 last，建立 SAM。
   (虽然要建 Trie 树，但具有一定的不可取代性，推荐学习。)

下面讲一下 GSAM 实现需要注意的一些细节

浅谈实现
实现 2

- 多个串按顺序 extend，当且仅当在 extend 一个新串的时候，将 last 指针设置到 root。
(非常好写，有细节需要注意，推荐学习。)

- 多个串按顺序 extend，当且仅当在 extend 一个新串的时候，将 last 指针设置到 root。
  (非常好写，有细节需要注意，推荐学习。)

首先先看第二种实现，网上很多人直接将单串 SAM 的代码原封不动搬来，然后按照上述处理当成 GSAM 用。虽然这种做法可以过很多题，但是是一种错误的写法。
因为这种写法会导致很多空节点（即 $len - link[len] = 0$）的产生*。为了消除空节点，我们要在原来 SAM 的基础上，进行如下特判。

* 虽然空节点并不影响时空复杂度，而且看似不会影响广义 SAM 的性质。但是网上有人说，空节点在维护某些信息的时候可能会出问题，如果有人感兴趣可以去研究研究。

浅谈实现
实现 2

- 多个串按顺序 extend，当且仅当在 extend 一个新串的时候，将 last 指针设置到 root。
(非常好写，有细节需要注意，推荐学习。)

浅谈实现
实现 2

- 多个串按顺序 extend，当且仅当在 extend 一个新串的时候，将 last 指针设置到 root。
  (非常好写，有细节需要注意，推荐学习。)

为了避免空节点的产生，代码中有两个地方需要特判。

1. 如果新增节点有完全一致的 endpos，不需要增加节点，只需要重置 last。
2. 如果新增节点已有范围更大的 endpos 需要分裂，则只需要创建一个新节点。

如果不理解的可以对照着我之前提供的代码进行理解。

浅谈实现
实现 3

- 对多个串建立 Trie 树，对 Trie 树进行 bfs 的同时改 last，建立 SAM。
(虽然要建 Trie 树，但具有一定的不可取代性，推荐学习。)

- 对多个串建立 Trie 树，对 Trie 树进行 bfs 的同时改 last，建立 SAM。
  (虽然要建 Trie 树，但具有一定的不可取代性，推荐学习。)

一般来说，上一种写法是够用的了。
但是有时候，题目多串的形式是通过 Trie 树的形式提供的（对 Trie 树有了解就会明白，这种情况不能保证 $\sum |S_i|$ 的大小）。
这个时候我们就需要对 Trie 树进行 bfs，同时每次 extend 动态调整 last。
(这里的广义 SAM 其实可以不用加实现 2 的特判，不过我习惯都加上)

- 对多个串建立 Trie 树，对 Trie 树进行 bfs 的同时改 last，建立 SAM。
  (虽然要建 Trie 树，但具有一定的不可取代性，推荐学习。)

一般来说，上一种写法是够用的了。
但是有时候，题目多串的形式是通过 Trie 树的形式提供的（对 Trie 树有了解就会明白，这种情况不能保证 $\sum |S_i|$ 的大小）。
这个时候我们就需要对 Trie 树进行 bfs，同时每次 extend 动态调整 last。
(这里的广义 SAM 其实可以不用加实现 2 的特判，不过我习惯都加上)

注意，遍历 Trie 树的时候一定要用 bfs!!! dfs 的时间复杂度是假的，下面给出 hack 例子和简要说明。

- 对多个串建立 Trie 树，对 Trie 树进行 bfs 的同时改 last，建立 SAM。
  (虽然要建 Trie 树，但具有一定的不可取代性，推荐学习。)

注意，遍历 Trie 树的时候一定要用 bfs!!!

- 对多个串建立 Trie 树，对 Trie 树进行 bfs 的同时改 last，建立 SAM。
  (虽然要建 Trie 树，但具有一定的不可取代性，推荐学习。)

注意，遍历 Trie 树的时候一定要用 bfs!!!

如果用 dfs 的话，那么遇到下面这组字符串建的 Trie 树。
{'ab','aab','aaab'...'a...ab'}
就会当场死亡，时间复杂度变成 $O(n^2)$。
大家可以自己模拟感受一下。

广义 SAM 复杂度 *

- 一个多串广义 SAM 的算法时空复杂度是 $O(|\sum| \times \sum |S_i|)$ 的。节点数上界 $2\sum |S_i|$。
- 一个给定 Trie 树建立广义 SAM 的时空复杂度是 $O(|\sum| \times |T|)$（其中 $|T|$ 为 Trie 树节点数）。节点数上界 $2|T|$。

为了方便起见，接下来的 $|T|$ 或 $\sum |S_i|$ 都统一用 n 表示。

注意广义 SAM 的节点和 SAM 的节点是完全一致的，所以 SAM 复杂度分析中的三种情况也分别适用于广义 SAM，大家自己类比，这里不过多赘述。

* 具体复杂度的证明可以参见本章末给出的拓展阅读资料。

复杂度介绍 Complexity

值得一提的是，广义 SAM（包括 SAM）在实际运用中有些操作并不是线性的，甚至 SAM 还经常与各种数据结构相结合来出题。所以不是只有线性题才可能是 SAM。

比如下面有个例题就将讲解一个非线性的广义 SAM 上的小 Trick。

- 【模板】广义后缀自动机（广义 SAM）(求多个串的本质不同子串)
- [Luogu]3346 [ZJOI2015] 诸神眷顾的幻想乡 (求多个 Trie 树的本质不同子串)
- [Luogu]3181 [HAOI2016] 找相同字符（求两个串个所有相同子串对的个数）

Count New String
[NowCoder]5699C

题目来源：2020 牛客多校第四场 C. Count New String

题目大意：给定一个字符集大小 10 的字符串 S，对于其每一个后缀 T，生成等长字符串 T'，使 $T'_i = \max(T_1..T_i)$。
求所有 T' 的本质不同子串个数。
$1 \leq |S| \leq 10^5$

比如'dbcad' 这个字符串，对于每一个后缀生成的 T' 集合为 {'ddddd','bccd','ccd','ad','d'}。

Count New String
[NowCoder]5699C

比如'dbcad' 这个字符串，对于每一个后缀生成的 T' 集合为 {'ddddd','bccd','ccd','ad','d'}。

我们好好观察一下上面这个例子，容易发现这些 T' 虽然不尽相同，但是似乎他们在某些情况下的部分后缀是相同的。

Count New String
[NowCoder]5699C

比如'dbcad' 这个字符串，对于每一个后缀生成的 T' 集合为 {'ddddd','bccd','ccd','ad','d'}。

我们好好观察一下上面这个例子，容易发现这些 T' 虽然不尽相同，但是似乎他们在某些情况下的部分后缀是相同的。

不妨我们将字符串 S 和题目设定翻转过来考虑这道题。于是我们顺水推舟就想到按照新的 S 前缀的变换来建 Trie 树。
而每增加一位 S_i，相当于在已建好的 Trie 树中最后一位比 S_i 大的节点后面开始续足够多个 S_i。

Count New String
[NowCoder]5699C

比如'dbcad' 这个字符串，对于每一个后缀生成的 T' 集合为 {'ddddd','bccd','ccd','ad','d'}。

我们好好观察一下上面这个例子，容易发现这些 T' 虽然不尽相同，但是似乎他们在某些情况下的部分后缀是相同的。

不妨我们将字符串 S 和题目设定翻转过来考虑这道题。于是我们顺水推舟就想到按照新的 S 前缀的变换来建 Trie 树。
而每增加一位 S_i，相当于在已建好的 Trie 树中最后一位比 S_i 大的节点后面开始续足够多个 S_i。

由于字符集大小为 10，所以另起炉灶的次数肯定小于等于 10 次。因此 Trie 树的节点数一定小于 $10|S|$。然后对 Trie 树建广义 SAM，求本质不同子串即可。

Count New String
[NowCoder]5699C

不过这道题其实不一定要真的把 Trie 树建出来。只需要用一个大小为 10 的数组存一下每一个字符在 Trie 树上最后一次出现对应的 SAM 节点就好了。然后一边“建”Trie，一边建广义 SAM。

Count New String
[NowCoder]5699C

不过这道题其实不一定要真的把 Trie 树建出来。只需要用一个大小为 10 的数组存一下每一个字符在 Trie 树上最后一次出现对应的 SAM 节点就好了。然后一边“建”Trie，一边建广义 SAM。

在现场的时候，有人因为 dfs 了 Trie 树来建广义 SAM 而被卡了。我构造了一下，的确如果数据为 $S =' ababababab...ab'$。dfs 就炸了。

Count New String
[NowCoder]5699C

不过这道题其实不一定要真的把 Trie 树建出来。只需要用一个大小为 10 的数组存一下每一个字符在 Trie 树上最后一次出现对应的 SAM 节点就好了。然后一边“建”Trie，一边建广义 SAM。

在现场的时候，有人因为 dfs 了 Trie 树来建广义 SAM 而被卡了。我构造了一下，的确如果数据为 $S =' ababababab...ab'$。dfs 就炸了。

这里可能就会有人有疑问了，一边建 Trie 一边建 SAM 的方法似乎也并不是严格的 bfs，为什么可以呢？这... 我也说不清楚，但是我的确构造不出来卡死这个方法的数据。
~~不如证明留给大家当课后习题吧~~。
(不过如果大家以后遇到类似题目，还是建 Trie 树之后再 bfs 最稳妥。)

All with Pairs
[NowCoder]5667A

题目来源：2020 牛客多校第二场 A. All with Pairs

题目大意：给定 n 个字符串。定义 $f(s_i, s_j)$ 为 s_i 的前缀与 s_j 后缀相同的最长长度，如果不存在则为 0。
求 $\sum_{i=1}^{n}\sum_{j=1}^{n} f(s_i, s_j)^2$ 对 998244353 取模。
$(1 \le n \le 10^5, 1 \le \sum s_i \le 10^6$, 字符串仅包含小写字母)

首先我们对字符串集合建广义 SAM，统计每个节点所代表的前缀是多少个串后缀。具体实现为每个节点设置一个 cnt，初始值赋值为 0，每个字符串的最后节点的 $cnt++$。最终每个节点 parent 树子树中 $\sum cnt$ 即代表对应前缀所可以匹配到的后缀的个数。

首先我们对字符串集合建广义 SAM，统计每个节点所代表的前缀是多少个串后缀。具体实现为每个节点设置一个 cnt，初始值赋值为 0，每个字符串的最后节点的 $cnt++$。最终每个节点 parent 树子树中 $\sum cnt$ 即代表对应前缀所可以匹配到的后缀的个数。

但是题目所求的之中，只有最长的需要算贡献。很容易发现，对于每个前缀而言，其 boarder 的贡献被重复计算了。所以我们应该对每个串前缀的 next 进行去重。也就是 $cnt[next]-=cnt$。

首先我们对字符串集合建广义 SAM，统计每个节点所代表的前缀是多少个串后缀。具体实现为每个节点设置一个 cnt，初始值赋值为 0，每个字符串的最后节点的 $cnt++$。最终每个节点 parent 树子树中 $\sum cnt$ 即代表对应前缀所可以匹配到的后缀的个数。

但是题目所求的之中，只有最长的需要算贡献。很容易发现，对于每个前缀而言，其 boarder 的贡献被重复计算了。所以我们应该对每个串前缀的 next 进行去重。也就是 $cnt[next]-=cnt$。

这里有个小 trick，通过打标记，是可以在建广义 SAM 的同时跑 KMP，并将 next 数组嵌入在广义 SAM 节点上的。

题目来源：[Adera 1 杯冬令营模拟赛] 串

题目大意：给定 n 个字符串，询问每个字符串有多少非空子串是所有 n 个字符串中至少 k 个的子串。

$1 \leq n, k, \leq 10^5, \sum |S_i| \leq 10^5$

先考虑一种暴力思维，很容易想到，如果我在建串的时候每个 SAM 节点都用一个 set 储存有该节点是多少个串的节点，那么在每个节点实际上的 set 就是对 parent 树上子树的 set 取并。

先考虑一种暴力思维，很容易想到，如果我在建串的时候每个 SAM 节点都用一个 set 储存有该节点是多少个串的节点，那么在每个节点实际上的 set 就是对 parent 树上子树的 set 取并。

这种算法的时空复杂度肯定不行，而且维护了不必要的信息，考虑是否可以只存 times（即在多少个串中出现）。这个时候我们考虑在建 SAM 的时候动态维护，即顺着 link 往下搜节点，使其 $times++$，为了避免重复，我们在建 SAM 的时候对每个 SAM 打标记，每个串保证不重复 $+1$ 即可。

先考虑一种暴力思维，很容易想到，如果我在建串的时候每个 SAM 节点都用一个 set 储存有该节点是多少个串的节点，那么在每个节点实际上的 set 就是对 parent 树上子树的 set 取并。

这种算法的时空复杂度肯定不行，而且维护了不必要的信息，考虑是否可以只存 times（即在多少个串中出现）。这个时候我们考虑在建 SAM 的时候动态维护，即顺着 link 往下搜节点，使其 $times++$，为了避免重复，我们在建 SAM 的时候对每个 SAM 打标记，每个串保证不重复 $+1$ 即可。

不过这个操作并非是广义 SAM 常规操作，最坏复杂度并不是线性的而是 $O(|S_i|\sqrt{|S_i|})$ 的 *。不过常数非常小而且需要非常特殊的串才能卡到上限。

先考虑一种暴力思维，很容易想到，如果我在建串的时候每个 SAM 节点都用一个 set 储存有该节点是多少个串的节点，那么在每个节点实际上的 set 就是对 parent 树上子树的 set 取并。

这种算法的时空复杂度肯定不行，而且维护了不必要的信息，考虑是否可以只存 times（即在多少个串中出现）。这个时候我们考虑在建 SAM 的时候动态维护，即顺着 link 往下搜节点，使其 $times++$，为了避免重复，我们在建 SAM 的时候对每个 SAM 打标记，每个串保证不重复 $+1$ 即可。

不过这个操作并非是广义 SAM 常规操作，最坏复杂度并不是线性的而是 $O(|S_i|\sqrt{|S_i|})$ 的 *。不过常数非常小而且需要非常特殊的串才能卡到上限。

* 这个操作复杂度证明和其他应用可以参考https://www.cnblogs.com/mangoyang/p/10155185.html

结尾 Ending

广义 SAM 在国内算法竞赛圈内第一次出现应该是在 2015 年的国家队刘研绎的论文里面。

而正是作为近几年新兴处理字符串的工具，广义 SAM 还是有很多待挖掘的地方，非常值得大家学习。

拓展阅读资料 Extension

- 后缀自动机在字典树上的拓展——刘研绎
- 广义后缀自动机——oi-wiki
- 后缀自动机及其应用——张天扬

感谢观看！