String

samjia2000

字符串

string

Hash

KMP&exKMP

Manacher

Trie

后缀数组

AC 自动机

SAM

后缀平衡树

其他算法

约定

对于一个字符串 s，用 n 表示 s 的串长， s[l:r] 表示 s[l],s[l+1]...s[r] 拼接而成的子串
border：如果存在一个字符串 t 同时是 s 的前缀也是 s 的后缀，那么 t 被称为 s 的 border

Hash

对于一个字符串 s ，以及一个质数 p 和模数 m ，定义
$hash(s,p,m)=sum(s[i] * p^i, i=1..n) \bmod m$
一般可以用来判断两个字符串是否相同
优点：容易维护

Hash

Hash- 例题 1

维护一个字符串，支持以下两种操作：

1. 在末尾加入一个字符
2. 给两个后缀求他们的 LCP

强制在线

n,q<=100000

Hash

维护 f[i][p] 表示 s[i:i+2^p-1] 的 hash 值
每次询问直接二分判断两个前缀是否相同即可。
时间复杂度 $O(q \log^2 n)$

Hash

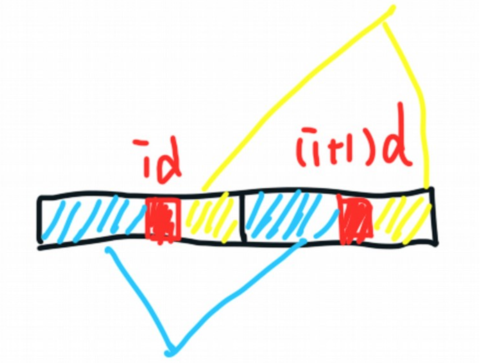
Hash- 例题 2

给出一个字符串 s，要求求出 s 中前一半与后一半相同的子串的数量。

n<=100000

Hash

假设某个合法的子串为 AA，其中 A 为一个长度为 d 的字符串。
枚举 d，记录 n/d 个关键点 d,2d,3d...
对于第 i 个关键点，利用二分 +hash 可以求出 s[id-d+1:id] 与 s[id+1:id+d] 的最长公共前缀以及最长公共后缀，记为 p[i] 和 q[i]。
考虑一个合法的子串，那么它一定经过了两个相邻关键点。
如图，红色点为关键点的位置，那么前后对应了两对相同的串，实际上对应了一组公共后缀以及公共前缀，利用 p[i] 和 q[i]，就可以计算出答案了。



KMP&exKMP

KMP 算法是拿来处理字符串匹配的。换句话说，给你两个字符串，你需要回答，B 串是否是 A 串的子串（A 串是否包含 B 串）。比如，字符串 A="I'm samjia2000"，字符串 B="samjia"，我们就说 B 是 A 的子串。

在 KMP 算法中，一个很关键的方法是，对于 i，有 next[i] 表示 max{k|k<i 且 s[1:k]=s[i-k+1:i]}，即 s[1:i] 中最长的既是前缀又是后缀的串(border)。

计算一个串的 next 数组的过程很简单，如下：

```
j = 0
for i = 2 to n :
    while (j>0) and (s[j+1]!=s[i]) : j=next[j]
    if (s[j+1]==s[i]) : j = j+1
    next[i]=j
```

KMP&exKMP

正确性?
首先如果 s[1:k]=s[i-k+1:i]，那么 s[1:k-1]=s[i-k:i-1]，也就是说 s[1:k-1] 是 s[1:next[i-1]] 的一段后缀以及前缀。
然后由于在不断跳 j 的过程中，实际上遍历了所有 s[1:i-1] 的 border，显然是会找到正确的 next[i] 的。
时间复杂度?
考虑到区间 [i-j+1,i]，实际上由于跳的过程中 j 在减少，右端 i 至多增加 n 次，j 也至多增加 n 次，那么总的时间复杂度是 O(n) 的。

KMP&exKMP

KMP&exKMP- 例题 1

有一棵有根树，每条边上有一个英文字符。（就是一棵 trie ）
有若干操作，每次会添加一个叶子或者给出一个节点，要求求出它到根的路径上形成的字符串的 border

$n<=100000$

KMP&exKMP

直接将 KMP 的 next 数组放到树上?
很遗憾由于 KMP 的复杂度是均摊分析的，这样实际上是有反例的。
比如说一开始有一个长度为 100000 的全是 'a' 的链，然后在最下面的节点上不断地加上若干个 'b' 的叶子，每次都要跳 100000 然后发现跳到了根节点。
正确的做法?
对每个节点除了维护 next 之外，还需要维护 to[x][c] 表示如果 x 的下一个字符是 c ，那么会跳到哪个节点上。
于是 to[x][c]=to[next[x]][c]
就可以 O(n∑) 的解决问题了（其中Σ为字符集大小）

KMP&exKMP

exKMP 算法可以计算出 f[i] 为 s 与 s[i:] 的 LCP，特殊的， f[1]=0
exKMP 的流程如下：

```
--------------------------------------------------------------------------------------------------
--------------
p=1
for i = 2 to n :
    if (p+f[p]-1>=i) : f[i]=min(f[i-p+1],p+f[p]-i)
    else : f[i]=0
    while (i+f[i]<=n and s[f[i]+1]==s[i+f[i]])f[i]=f[i]+1
    if (i+f[i]>p+f[p]) : p=i
--------------------------------------------------------------------------------------------------
--------------
```

时间复杂度 O(n)

KMP&exKMP

```
p=1
for i = 2 to n :
    if (p+f[p]-1>=i) : f[i]=min(f[i-p+1],p+f[p]-i)
    else : f[i]=0
    while (i+f[i]<=n and s[f[i]+1]==s[i+f[i]])f[i]=f[i]+1
    if (i+f[i]>p+f[p]) : p=i
```

正确性?

分析一下 f 的过程即可知道正确性。

KMP&exKMP

```
-----------------------------------------------------------------------
----------
p=1
for i = 2 to n :
    if (p+f[p]-1>=i) : f[i]=min(f[i-p+1],p+f[p]-i)
    else : f[i]=0
    while (i+f[i]<=n and s[f[i]+1]==s[i+f[i]])f[i]=f[i]+1
    if (i+f[i]>p+f[p]) : p=i
-----------------------------------------------------------------------
----------
```

时间复杂度?

因为 p+f[p] 是递增的，又由于 f[i]=min(f[i-p+1],p+f[p]-i) 的存在， while 执行的次数是 O(n) 的

故总的时间复杂度是 O(n)

Manacher

Manacher 是用来求一个字符串内的回文串的算法。
为了避免奇偶性的讨论， Manacher 首先从字符串 s 构造一个新的字符串 t，其中 |t|=2|s|+1
如果 i 是奇数，那么 t[i]='#'(实际上是一个不在字符集里面的字符即可)，否则 t[i]=s[i/2]
记 f[i] 表示 t 中以 t[i] 为中心的奇回文串的数量。
Manacher 的流程如下：

```
---------------------------------------------------------------------------------
p=0
for i = 1 to 2n-1 :
    if (p+f[p]-1>=i) : f[i]=min(p+f[p]-i,f[p*2-i])
    else : f[i]=0
    while (i+f[i]<=2n-1 and i-f[i]>0 and t[i+f[i]]==t[i-f[i]]): f[i]=f[i]+1
```

Manacher

```
-----------------------------------------------------------------------------------------
p=0
for i = 1 to 2n-1 :
    if (p+f[p]-1>=i) : f[i]=min(p+f[p]-i,f[p*2-i])
    else : f[i]=0
    while (i+f[i]<=2n-1 and i-f[i]>0 and t[i+f[i]]==t[i-f[i]]): f[i]=f[i]+1
    if (i+f[i]>p+f[p]) : p=i
-----------------------------------------------------------------------------------------
```

可以发现 Manacher 算法的过程中也用到了 p，那么跟 exKMP 中的 p 的作用其实是类似的，由于 p+f[p] 是不下降的，那么保证了时间复杂度。

Manacher

Manacher- 例题 1

给出一个字符串 s，求 s 的最长双回文子串的长度，一个字符串是双回文串当且仅当他可以被分成两个串 AB 使得 A 和 B 都是回文串。

n<=100000

Manacher

使用 Manacher 求出 f 之后，枚举双回文串的分界点 p(t[p]='#')，然后对于左边的回文串，假设中心是 x，那么要找的是满足 x+f[x]-1>=p-1 的最小的 x，对应的找到右边最大的 y

可以 O(n) 求出 x 和 y

Trie

一种树形结构，用于保存大量的字符串。
利用字符串的公共前缀节省空间，操作简单方便，应用广。
但每个节点的空指针会占用大量内存。当字符集较大时很难同时兼顾时间和空间，这种时候可以使用 map 等数据结构来存储边。
应用：存储字符串，前缀匹配，字符串出现次数等。
并且支持可持久化!

AC 自动机

~~AC 自动机是一个写了就可以让你自动 AC 的算法~~
AC 自动机的构造是基于 Trie 的
将所有模板串建成一棵 Trie，在 Trie 上跑 KMP

和 KMP 算法一样，需要对 Trie 上每一个点 x 建立 fail 指针，使得指向的点 y 代表的串，为当前点 x 的最长后缀。

匹配的时候用文本串在 Trie 上跑，失配了就沿着 fail 指针往之前跳。

AC 自动机

AC 自动机 - 例题 1

现在有 m 个字符集为小写字母的模板串 s[1],s[2]..s[m]，随机一个长度为 k 的只有小写字母的串，问模板串在这个随机串中作为子串出现的次数的期望值。

模板串的长度总和不超过 5000
k<=5000

AC 自动机

建出 AC 自动机之后，设 f[i][x] 表示长度为 i 的随机串最后匹配到 AC 自动机中的 x 号节点模板串作为子串出现的期望次数， g[i][x] 则表示对应的概率。
对于每个节点 z ，计算出其代表的字符串是模板串的后缀的个数 cnt[z] 。
转移的时候考虑在这个串后面新增一个字符 c ，然后假设节点 x 后添加 c 会跳到 y ，那么就是：
g[i][x] --> g[i+1][y]
f[i][x] + g[i][x]*cnt[y] --> f[i+1][y]

后缀数组

后缀数组可以对一个字符串 s 求出其所有后缀的排名， SA[i] 表示排第 i 名的后缀， Rank[i] 则表示 s[i:] 在 SA 中的排名。

一般后缀排序有两种方法，倍增和 DC3 ，倍增是应用较广的，也是更为实用的
~~（不会 DC3 瑟瑟发抖）~~

设 suf(i) 表示第 i 个位置开头的后缀。

对于每个 suf(i) ，定义它长度为 k 的前缀为 suf(i,k)

后缀数组

对于每个 $suf(i)$，定义它长度为 k 的前缀为 $suf(i,k)$
显然有几个性质

$suf(i,2k)<suf(j,2k)$ 当且仅当 $suf(i,k)<suf(j,k)$ 或 $suf(i,k)=suf(j,k)$ 并且 $suf(i+k,k)<suf(j+k,k)$
$suf(i,2k)=suf(j,2k)$ 当且仅当 $suf(i,k)=suf(j,k)$ 并且 $suf(i+k,k)=suf(j+k,k)$

设数组 SA_k[],Rank_k[] 表示在只排序每个后缀长度为 k 的前缀条件下的 SA,Rank
那么完全可以利用 Rank 来比较大小

后缀数组

那么完全可以利用 Rank 来比较大小

然而，在 k 长度前缀下可能有完全相同的字符串，那么他们的 Rank_k 就是相等的

于是每个 suf(i,2k), 都有两个关键字 Rank_k(i),Rank_k(i+k) ，可以利用双关键字基数排序 O(N) 求出 SA_2k[],Rank_2k[], 再到 4k,8k,16k ，最后当 k 大于等于 N 了， SA_k 和 Rank_k 就是 SA 和 Rank 了。那么排序就结束了（显然超出部分并不影响）

当然，初始时你需要对每一个字符做一遍排序，相当于 k=1

后缀数组

排序的部分，使用的是基数排序
我们发现 rank 数组的范围是 |S| 以内的。
首先求出按第二关键字排序好的数组 s2[]
然后按第一关键字排序（如果第一关键字相同，则它在 s2[] 中的先后顺序就是按第二关键字排序的结果）

通过计算好的新的 SA_k 来计算 Rank_k

结合代码来理解一下

后缀数组

Rank[i] 表示第 i 个后缀的排名， sa[i] 表示排名为 i 的后缀的开头

求出 SA[] 以及 Rank[] 之后，还需要求 height[]，其中 height[i] 表示 s[sa[i]:] 与 s[sa[i-1]:] 的 LCP.

为了方便，记 ps(i) 满足 Rank[ps(i)]=Rank[i]-1

那么 height[i-1]-1=LCP(s[i-1:],s[ps(i-1):])-1<=LCP(s[i:],s[ps(i-1)+1:])<=LCP(s[i:],s[ps(i):]=height[i]
即 height[i-1]-1<=height[i]
那么按照 i=1..n 的顺序计算 height，时间复杂度为 O(n)

求出 height 之后，当我们想求两个后缀 s[x:] 和 s[y:] 的 LCP 的时候（Rank[x]<Rank[y]），只需要求 height[Rank[x]+1..Rank[y]] 中的最小值即可。
使用 RMQ 进行维护即可。

后缀数组

1、求任意两后缀 LCP （方法前面讲过）
2、求出现过至少两次的最长子串（可以重叠）
子串就是后缀的前缀
相当于求任意两后缀最长 LCP，也就是 height 的最大值
3、求出现过至少两次的子串个数
同上面，在 height 上考虑，每次答案当然应该加上 height[i]，
但是要减掉前面已经算过的
所以答案每次加上 max(0,height[i]-height[i-1])
4、求本质不同的子串个数
容易发现后缀 SA[i] 有 n-SA[i]+1 个前缀
其中有 height[i] 个和之前重复
答案累加 n-SA[i]+1+height[i] 即可

SAM

有限状态自动机?
有限状态自动机的功能是识别字符串，令有一个自动机 A，如果他能识别一个字符串 str，那么 A(str)=true，不然 =false。
自动机的五个重要部分：alpha：字符集；state：状态集合；init，初始状态集合（如 AC 自动机的根节点）；end：终止状态集合；trans，状态转移函数。
终止状态，既可以是成功识别，也可以是识别失败。
我们记 trans(x,c) 为当前状态为 x，读入字符 c 后，转移到的状态。记不存在的状态为 null。SAM 的 init 集合只有一个元素，我们记为状态 0。
为了方便，记 trans(x,str) 为从状态 x 把整个字符串都读入了转移到的状态。
相当于：

```
for i=1 to |str|
    x=trans(x,str[i]);
```

SAM

~~Sam 是一名菜鸡选手~~
SAM，即后缀自动机，是有限状态自动机的一种，可以识别 s 的所有后缀，实际上也可是识别 s 的所有子串。
SAM 的构造及其证明可以在各种网上教程或是课件中查看，由于太过冗长，主要介绍 SAM 的重要性质以及需要注意的问题。

SAM

SAM 中的每个节点 x 代表着一类字符串 T(x)，记其中最长的为 t(x)（t(x) 一定是 s 的子串），每个节点有一个 right(x) 集合， right(x)={r| s[r-|t(x)+1|:r]=t(x)}，即 t(x) 在 s 中的出现位置， T(x) 中的字符串的 right 集合均相同

所有节点组成了一个树状结构（被称为 parent 树），节点 x 的父亲为 t(y) 最长且 t(y) 是 t(x) 的后缀的节点 y，实际上可以看成节点 x 表示了 t(x) 中长度为 |t(y)|+1..|t(x)| 的后缀

所有节点还组成了一个有向图的结构，每条边上有一个字符，一条边 (u,v,c)，表示 t(u)+c 得到的字符串可以被节点 v 接收。

SAM

```
struct SAM {
	int go[26], fail, len;
} tr[MAXN * 2];
void add(int c) {
	int np = ++ tot, p = last;
	tr[np].len = tr[p].len + 1, last = np;
	tr[np].size = 1;
	for (;p && !tr[p].go[c]; p = tr[p].fail) tr[p].go[c] = np;
	if (!p) tr[np].fail = root; else {
		int q = tr[p].go[c];
		if (tr[p].len + 1 == tr[q].len) tr[np].fail = q; else {
			int nq = ++ tot;
			tr[nq] = tr[q];
			tr[q].fail = tr[np].fail = nq;
			tr[nq].len = tr[p].len + 1, tr[nq].size = 0;
			for (; p && tr[p].go[c] == q; p = tr[p].fail) tr[p].go[c] = nq;
		}
	}
}
```

SAM

状态数的证明?
可以考虑一个比较粗暴的证明，由于 SAM 的 parent 树实际上是 s 的所有子串组成的 Trie 的缩减版， parent 树是保留了所有叶子以及儿子至少两个的节点组成的树，故节点数不超过 2n-1

时间复杂度证明?
见陈立杰的论文

SAM

在有的题目中，会遇到需要匹配一堆模式串的子串的情况。
第一个做法是将所有模式串连接在一起，中间加上一个不在字符集内的字符，然后对得到的大的串建 SAM
第二个做法是先建出 Trie，然后在 Trie 上按照 bfs 的顺序建 SAM （为什么要 bfs，见论文《后缀自动机在字典树上的拓展》）

后缀平衡树

后缀平衡树实际上是用平衡树来维护后缀数组。
对于一个串 s ，考虑从后往前加入每个后缀，每次加入 s[i:] 的时候，当需要比较 s[i:] 和 s[j:] 的时候，需要先比较 s[i] 和 s[j] ，如果 s[i] 和 s[j] 相同，那么需要比较 s[i+1:] 和 s[j+1:] 的大小关系。
注意每次插入 s[i:] 的时候，在平衡树上需要 O(log n) 次比较，如果每一次都需要再比较 s[i+1:] 和 s[j+1:] ，那么需要 $O(\log^2 n)$ 的时间复杂度。
使用 treap 作为我们的数据结构之后，对每个节点 x 的子树定一个权值区间 $[l_x,r_x]$ ， x 的权值记为 $(l_x+r_x)/2$ ，保证左子树的权值小于 x 的权值小于右子树的权值，然后比较的时候比较权值即可。
旋转的时候重构权值，期望的重构复杂度是 O(n log n)
在大部分情况下，后缀数组可以替代后缀平衡树的功能。

其他算法

FFT
在需要通配符匹配的时候，可以使用 FFT 进行加速来计算。

比如说，有两个串 s 和 t， s 中的一个子串和 t 可以匹配，当每个对应位字符的差的绝对值不超过 1，问有哪些位置可以与 t 匹配。

考虑两个字符的差 d， d=-1/0/1 的时候可以匹配，那么由于 d(d+1)(d-1)=d^3-d 当 d=-1/0/1 的时候为 0，而其他时候大于 0，那么可以用 FFT 计算对应位的差的立方和然后就可以判断是否匹配了。

其他算法

回文自动机 / 回文树
回文自动机可以识别一个串的所有回文子串。

考虑一个 DFA 能识别 S 的所有回文子串 .

- 为了方便设立两个根 R1, R2，长度分别是 0 和 -1.
- trans[x][c] 表示 x 左右两端加上 c 以后的字符串 .
- fa[i] 是去掉两边字符后的回文串 .
- fail 定义为 x 的最长回文真后缀 .
- fail[R2] = fail[R1] = R2.

其他算法

增量构造 .

- 假设有 PAM(S)，构造 PAM(Sc)
- 实时维护最长回文后缀 .
- 每次暴力往后缀的 fail 跳直到 前一个字符也是 c.
- 然后更新一下最长回文后缀 .
- 如果这个串是一个新的串，那么继续往前跳找下一个 c.
- 均摊复杂度 O(n)

HELL
O

We are s
different
but are from the
same world.

CSP2019