BWT变换——数据整理

一. 简述

BWT变换（百度百科翻译为BWT压缩算法）是一种对字符串的转换算法，可用于数据压缩，DNA测序（？），模糊匹配。

在数据压缩中，通常希望被压缩的数据具有一些良好的性质，比如多个相同的字符连在一起，BWT算法则是将字符串通过某种变换成为一个具有这样良好性质的新字符串，这种变换当然是可逆的，可以通过逆变换求解。

二. 朴素BWT变换

首先，你得到了一个原始字符串：abcacabdc，字符串长度为len-1，以下是算法流程：

1. 为了更容易识别字符串的首位，我们在字符串末尾加一个特殊符号（字符串中不包含的符号）作为标记，处理后的字符串为abcacabdc#。
2. 将原始串向右移动一位，可以生成一个新的字符串：#abcacabdc。
3. 如此，不断向右移，一共可以生成len个不同的字符串

```
1   abcacabdc#
2   #abcacabdc
3   c#abcacabd
4   dc#abcacab
5   bdc#abcaca
6   abdc#abcac
7   cabdc#abca
8   acabdc#abc
9   cacabdc#ab
10  bcacabdc#a
```

4. 将这些字符串按照字典序排序可得：

```
1   #abcacabdc
2   abcacabdc#
3   abdc#abcac
4   acabdc#abc
5   bcacabdc#a
6   bdc#abcaca
7   c#abcacabd
8   cabdc#abca
9   cacabdc#ab
10  dc#abcacab
```

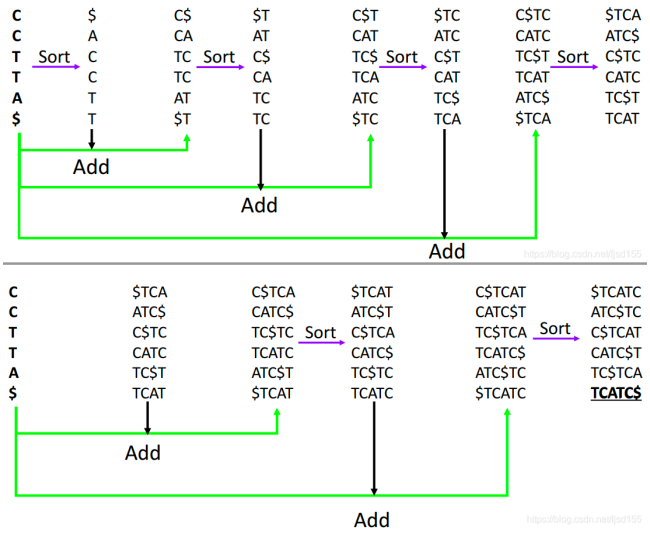
5. 将这个字符串矩阵的最后一列按顺序抄写下来，作为BWT变换的结果，即：c#ccaadabb

朴素BWT变换算法时间复杂度O(n^2logn)。

三. 朴素BWT逆变换

如何将BWT字符串还原为原始串？

从最后一列开始，尝试还原整个矩阵。设最后一列为BWT串，建立k个空的字符串设为当前串。接下来循环n次，每次将当前串按字典序排序，并将BWT串逐位加在当前串前面，如图：



正确性：我也不知道…

时间复杂度O(n^2logn)

四. 后缀数组优化BWT变换

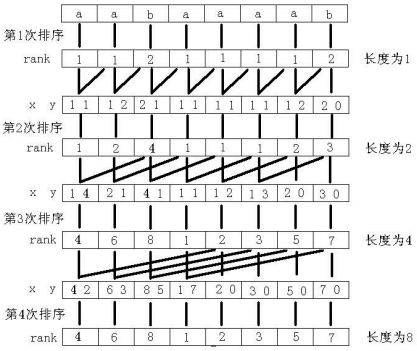
虽然上述算法清晰的阐释了变换过程和原理，但是运行效率不能令人满意。

这里提出一种基于后缀数组的优化算法。

考虑到在BWT矩阵中，每行第一个字符的前一个字符是该行的最后一个字符。而矩阵又是由该字符串的所有循环同构按照字典序排序得到，所以可以使用后缀数组快速的求出每行第一个字符所在原字符串中的位置，该位置之前一个字符为改行最后一个字符。通过SA求BWT的公式如下：

$$BWT[i] = \begin{cases} T[SA[i]-1], & \text{if } SA[i] > 0 \\ \$, & \text{otherwise} \end{cases}$$

由于后缀数组可以使用DC3算法线性求解或倍增算法O(nlogn)求解，所以该优化算法可以达到的时间复杂度为O(n)。（附，后缀数组倍增算法图）



五. BWT逆变换

同样，我们也希望BWT逆变换有更高的计算效率，BWT逆变换的优化相对简单。

对于BWT矩阵（左），我们现在只知道如右图的两列，最后一列是BWT串，第一列可以通过BWT串排序得到。

#abcacabdc	#.....c
abcacabdc#	a.....#
abdc#abcac	a.....c
acabdc#abc	a.....c
bcacabdc#a	b.....a
bdc#abcaca	b.....a
c#abcacabd	c.....d
cabdc#abca	c.....a
cacabdc#ab	c.....b
dc#abcacab	d.....b

观察可以发现，该矩阵具有三个规律：1.每行第一个字符的前一个字符是该行最后一个字符。2.第一列中相同的字母相对位置与最后一列一样。如字母c，第一个字母c的下标在第一列和最后一列都是8，第二个c的下标都是4，第三个c的下标都是2。3.第一行最后一个字符是原字符串倒数第二个字符（倒数第一个字符是添加的占位符）

利用这些规律，我们可以倒着复原原字符串，先从第一行最后一列找到c，作为原字符串最后一个字符，在找到左边对应的第7行，在第7行最后一列找到c的前一个字符是d…以此类推，可以线性完成BWT逆变换，所以BWT逆变换的时间复杂度被优化为O(n)。

六. 查找子串

如何在不还原BWT串的情况下查找子串？利用与优化逆变换的相同思路进行。在下图中，需要查找aac这个子串，首先在第一列找到c所在的区间，再看其前一个字符是否为a，并通过与逆变换相同的思想进行状态转移，图片清晰易懂，不做赘述。

对图片的一些解释：当前匹配串在左侧一定是连续的区间，但是在右侧可能不连续，所以时间复杂度为O(n^2)（或许有更好的做法？目前没有想到）