1 Problem

For a string A of length n, there are n cyclic rotations of A. For example, if the string is A = abba, the 4 cyclic rotations are: abba, bbaa, baab, aabx. If we lexicographically order those four strings, they are ordered aabx, abba, baab, bbaa. The last character of each string, in this lexicographic string order, concatenated together creates the string A' = xaba. In general, given string A, create the string A' of length n by forming the n cyclic left rotations of A, sorting them in lexicographic order, extracting and concatenating the last character of each of the strings, in sorted order. Index I points to the position in A' occupied by the last character of the original string. In the example, I = 2. Let (A',I) denote the encoded string and the pointer to the character contributed by the original string. So the encoding of abba is (A',I) = (xaba, 2).

Problem: Devise an efficient algorithm to reconstruct A from (A',I) in general. Describe an efficient algorithm, using a suffix tree or array to create the encoding (A',I).

The transform is a widely-used method called the Burrows Wheeler Transform. It is often used on strings before using other compression methods: (A',I) is compressed rather than the original string A. Some compression methods are typically able to better compress (A',I) than A. This depends of course on what A is and the compression method used.

2 Answer: BWT - inversion method

Here is a method to do the inversion of the BWT (assume the normal English alphabet). Let A be the original string, and let A' be the string produced by the BW transform. Think of the BWT as a compact representation of the lexicographically sorted list of left shifts of A. Let M denote the lexicographically sorted list of shifts of A. Although the BWT does not explicitly produce M, we can reason about it.

Method

1. Survey the characters in A', i.e., the characters in the string, to learn the number of a's, b's, etc. Therefore we know the complete partition of M based on the first character of the strings in M. For example, suppose 'a' occurs 10 times in A. That means that the first 10 characters in A' are the

last characters in the strings that begin with ‘a’. In general, for any character c in A', we can figure out the last characters in the strings that begin with c.

2. In the BWT we also have a pointer to the location in A', and hence into M, of the original string A, i.e., the null shift of A. Suppose it is in position i in the section of the partition of A' for character denoted c, and let the character in A' in that position be denoted x. That means that the original string starts with c and ends with x, but we won’t immediately use that fact.

We can deduce that the original string A starts with c and is the i’th smallest string that starts with c. Now consider the left shifts of A that put character c at the end. A left shift of A by one position is one of those strings. Denote that string by A''. It follows that A'' is the i’th smallest string in M that ends in c. Therefore, we can find the entry for A'' in A' by finding the i’th occurrence of character c in A'. Suppose we find it in the j’th position in the section of the partition for character c'. It follows that A starts with cc'. It also follows that A is the j’th smallest string in M that ends with c'. So, now we look for the j’th occurrence of character c' in A' to find the third character of A etc.

What is the running time of this? With the proper data structures, it can be done in linear time. The repeated task is to answer the question: Given a character c and a position i, what part of the partition of A' is the i’th occurrence of c in? To be able to answer these queries repeatedly, when we survey A' from left to right, we build up an array for each character, noting the part of the partition of A' that each character falls into. With those arrays, we do each task in constant time, so $O(n)$ overall.

3 Another method

The above might just be a roundabout, but more efficient, way to describe the better-known method:

0. Think of A’ as an n by 1 column vector, denoted by M.

Repeat n times:

1. Lexicographically (but stably, meaning that ties are broken based on the relative order in M of any ties) sort the rows (strings) in M to obtain a new matrix M'.

2. Prepend a copy of A' to M', assigning the resulting matrix to M.

This essentially implements radix sort on the set of n rotated strings, even though we don't know the full M. But it does this in a somewhat subtle way. In radix sort, we work from the rightmost column to the left. So when doing successive stable sorts on M, we are essentially sorting the strings after they have been circularly shifted right again by $n-1$ places, so that the character at the start of the BWT list is now at the end. But how do you make this linear time?

4 Problems (should you wish to solve them)

1) Show that if we don't know the pointer to the null shift, then we cannot invert the BW string to obtain the original string.

2) Show that if we know only the first column, not the last column, then we cannot invert - don't just show that the inversion algorithm fails - explain why no algorithm could work.

3) Show that if we don't know the pointer to the null shift, we can still deduce a circular shift of the original string.