Burrows Wheeler Transformation and its Applications

Khoong Wei Hao

Department of Mathematics
National University of Singapore

July 28, 2017

1. Introduction
   - History
   - Data Compression
     - Lossy Compression
     - Lossless Compression
   - Suffix, Suffix Array, and Suffix Trees
2. The Burrows Wheeler Transform
   - The Reversible Transformation
   - Effectiveness of the String Compression
   - An Efficient Implementation
   - FM Index
     - Backwards Matching
3. Applications of the Burrows Wheeler Transform
   - Implementation of the Transform
   - Performance of the Implementation
     - Compression for Binary Data
     - Compression for Letter-Based Texts

4 De Bruijn Sequences in the BWT
- Preliminaries
- De Bruijn Graphs
- De Bruijn Sequences in the Inverse BWT

5 Variants of the Burrows Wheeler Transform
- Bijective Variant of the BWT

6 Conclusion
- Selected References

History of the Burrows Wheeler Transform (BWT)

The Burrows-Wheeler Transform was invented by Michael Burrows and David Wheeler in 1994, while Burrows was working at DEC Systems Research Center in Palo Alto, California. The algorithm is based on a once unpublished work by David Wheeler in 1983, while he was working at AT&T Bell Laboratories.

Lossy Compression

In data compression, lossy compression involves permanently eliminating certain information in the data file, especially redundant information, to reduce the file size when compressed.

When the file is decompressed, only a portion of the original information will be present, although the difference from the original is not entirely noticeable.

Lossless Compression

Lossless compression methods allow the original data to be reconstructed from the compressed data exactly. In other words, lossless compression reduces the file size without degrading the quality of the original data (i.e. images).

Definition 1.3.1.

An alphabet is denoted by $\sum$, a finite set of characters or symbols.

Definition 1.3.2.

A string is a finite sequence of character or symbols from an alphabet $\sum$, enclosed by quotes ‘ ’ or “ ”. Denote $\sum^*$ as the set of all possible strings over an alphabet $\sum$.

Example

Given $\sum = \{a, b\}$, we have the finite set $\sum^* =$ {‘ ’, ‘a’, ‘b’, ‘aa’, ‘ab’, ‘ba’, ‘bb’, ‘aaa’, . . . }, and any element of $\sum^*$ is a possible string.

Definition 1.3.3.

A substring of a string T is a string T' that is a sequence of consecutive characters from T. A proper substring of T is any substring S, such that $S \neq T$.

Example

For example, 'hello' is a substring of 'hello world'.

Suffix, Suffix Array, and Suffix Trees III

Definition 1.3.4.

A prefix of a string T is a substring of T that begins with the first character of T. Formally, $\tilde{S}$ is a prefix of $T \iff \exists V \in \sum^*$ such that $T = \tilde{S}V$. A proper prefix of T is not equal to T.

Definition 1.3.5.

A suffix of a string T is any substring of T that includes the last character. Formally, a string S is a suffix of $T \iff \exists V \in \sum^*$ such that $T = VS$. A proper suffix of T is not equal to T $(i.e \sum^* \neq \emptyset)$.

Definition 1.3.6.

Let $T = T[0]T[1]\dots T[n-1]$ be a string of n characters, and let T[i,j] denote the substring of T ranging from i to j. We define the suffix array SA_T of T to be the array of integers $[0, n-1]$ that contains the starting positions of suffixes in lexicographical order, where $SA_T[i]$ contains the starting position of the i-th smallest suffix in T, and $T[SA_T[i-1], n] \leq T[SA_T[i], n] \; \forall \; 0 < i \leq n-1$.

Definition 1.3.8.

A string t is a cyclic rotation (or conjugate) of a string s if $t[0..n-1] = s[i..n-1]s[0..i-1]$ for some $0 \leq i \leq n-1$.

Algorithm A - Burrows-Wheeler Transform (BWT)

Let T be an input string of n characters $T[0], T[1], \ldots, T[n-1]$ selected from an ordered alphabet $\sum$ of the characters. We illustrate the method by an example as follows: Let T=‘abraca’ be a string, where $n = 6$ and alphabet $\sum = \{$‘a’,‘b’,‘c’,‘r’$\}$. For example, we have for $n = 6$, $T[0] =$ ‘a’, $T[1] =$ ‘b’, $T[2] =$ ‘r’, $T[3] =$ ‘a’, $T[4] =$ ‘c’, $T[5] =$ ‘a’. Next, we construct $n = 6$ strings (rotations) S_0, S_1, $\ldots$, $S_5 (= S_{n-1})$ such that

$$\begin{aligned} S_0 &= T[0] \ldots T[n-1] = \text{‘abraca’} \\ S_1 &= T[1] \ldots T[n-1] T[0] = \text{‘bracaa’} \\ S_2 &= T[2] \ldots T[n-1] T[0] T[1] = \text{‘racaab’} \\ &\ldots \\ S_5 &= T[n-1] T[0] \ldots T[n-2] = \text{‘aabrac’} \end{aligned} .$$

Algorithm A (Continued)

The next step is to sort $S_0, \ldots, S_5 (= S_{n-1})$ lexicographically. So from the string T, we have the sorted rotations:

$$\begin{aligned} S_5 &= \text{`aabrac'} \\ S_0 &= \text{`abraca'} \\ S_3 &= \text{`acaabr'} \\ S_1 &= \text{`bracaa'} \\ S_4 &= \text{`caabra'} \\ S_2 &= \text{`racaab'} \end{aligned}$$

Note that at least one of the strings S_i, $0 \leq i \leq 5 (= n-1)$ contains the original string T. The above outputs from the sorted rotations can also be represented by a $n \times n$ matrix M, whose elements are the characters $T[0], T[1], \ldots, T[n-1]$, and rows are the rotations (cyclic shifts) of T, sorted in a lexicographical order.

Algorithm A (Continued)

Denote I as the index of the first row of matrix M that contains the original string S. In this example, index $I = 1$, and matrix M given by

row	
0	aabrac
1	abraca
2	acaabr
3	bracaa
4	caabra
5	racaab

Let L be the output string of the transform which consists of the last character in each of the rotations in their sorted order. For e.g., L is the last column of M, and $L[0] = M[0, n-1], L[1] = M[1, n-1], \ldots, L[n-1] = M[n-1, n-1]$. The output of the transform is the ordered pair (L, I). Here, we have $L =$ 'caraab' and $I = 1$.

T-ranking I

Definition 2.1.1. (T-ranking)

Give each character in T a rank, equal to the number of times the character occurred previously in T.

Example

Let T=‘*abraca*’, and we re-write it as ‘$a_0 b_0 r_0 a_1 c_0 a_2$’. Re-writing matrix M, we have

$$M = \begin{bmatrix} a_2 & a_0 & b_0 & r_0 & a_1 & c_0 \\ a_0 & b_0 & r_0 & a_1 & c_0 & a_2 \\ a_1 & c_0 & a_2 & a_0 & b_0 & r_0 \\ b_0 & r_0 & a_1 & c_0 & a_2 & a_0 \\ c_0 & a_2 & a_0 & b_0 & r_0 & a_1 \\ r_0 & a_1 & c_0 & a_2 & a_0 & b_0 \end{bmatrix} \qquad (\star)$$

LF Mapping

Definition 2.1.2. (LF Mapping)

Let L and F denote the last and first columns of the matrix M obtained by Algorithm A respectively. Then the i^{th} occurrence of a character c in L and the i^{th} occurrence of c in F corresponds to the same occurrence in the original string T.

Algorithm B - Reverse Transform

Let L be the string consisting of the last characters of the sorted rotations $S_0, \ldots, S_{n-1}$ and I, which denotes the position of position of S_0 in L. The reverse transform will yield the original string T, of length n.

Firstly, we find the first character of each rotation S_i. Let F be the first column of the matrix M in Algorithm A, where as in Figure 2.1, we define M to be:

$$M = \begin{bmatrix} a & a & b & r & a & c \\ a & b & r & a & c & a \\ a & c & a & a & b & r \\ b & r & a & c & a & a \\ c & a & a & b & r & a \\ r & a & c & a & a & b \end{bmatrix}$$

Algorithm B (Continued)

To get F, we sort the characters of L. From the example in Algorithm A and matrix M above, we have $F =$ 'aaabcr'.

Next, given F and L, we need to determine which character should come after a certain character in F.

To help us determine the order of the characters above, we first re-write M where each character in T='*abraca*' has a rank, where we re-write it as '$a_0 b_0 r_0 a_1 c_0 a_2$'.

Algorithm B (Continued)

Re-writing matrix M, we have

$$M = \begin{bmatrix} a_2 & a_0 & b_0 & r_0 & a_1 & c_0 \\ a_0 & b_0 & r_0 & a_1 & c_0 & a_2 \\ a_1 & c_0 & a_2 & a_0 & b_0 & r_0 \\ b_0 & r_0 & a_1 & c_0 & a_2 & a_0 \\ c_0 & a_2 & a_0 & b_0 & r_0 & a_1 \\ r_0 & a_1 & c_0 & a_2 & a_0 & b_0 \end{bmatrix} \qquad (\star)$$

Looking down columns F and L, we observe that the the a_i's occur in the order: a_2, a_0, a_1. In fact, this holds true for any other character. This is a case of last-to-first column (LF) mapping.

Algorithm B - Reverse Transform IV

Algorithm B (Continued)

Now, let M' be the matrix obtained by rotating all the rows of M one character to the right, such that for each $i = 0, \ldots, n-1$, and each $j = 0, \ldots, n-1$,

$$M'[i,j] = M[i,(j-1) \; mod \; n],$$

where the first column of M' equals to the last column of M. For example, from $(\star)$, we have

$$M' = \begin{bmatrix} c_0 & a_2 & a_0 & b_0 & r_0 & a_1 \\ a_2 & a_0 & b_0 & r_0 & a_1 & c_0 \\ r_0 & a_1 & c_0 & a_2 & a_0 & b_0 \\ a_0 & b_0 & r_0 & a_1 & c_0 & a_2 \\ a_1 & c_0 & a_2 & a_0 & b_0 & r_0 \\ b_0 & r_0 & a_1 & c_0 & a_2 & a_0 \end{bmatrix} \qquad (\star\star)$$

Algorithm B - Reverse Transform V

Algorithm B (Continued)

Now, using F and L, the first columns of matrices M and M' respectively, we compute a vector V (an array in a programming context) such that row j of M' corresponds to row $V[j]$ of M.

Note that in Algorithm A, index I is defined in a way that row I of M is the original string T. Hence, the last character of T is $L[I]$.

Next, we use V to derive the predecessors of each character by using $T[n-1-i] = L[T^i[I]]$ for each $i = 0, \dots, n-1$, where $V^0[y] = y$, and $V^{i+1}[y] = V[V^i[y]]$. From this, we get T, the original input string for the compression transform.

Algorithm B - Reverse Transform VI

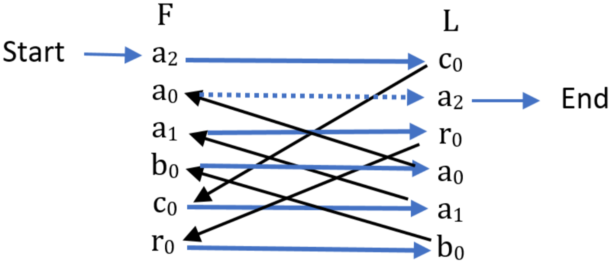


Figure: Reverse BWT starting at the right-hand-side of T and moving left

Consider the string ‘tomorrow and tomorrow and tomorrow’. Then by Algorithm A and the function bwt in Python (see Appendix), we obtain the output:

```
>>> bwt('tomorrow and tomorrow and tomorrow')
'wwdd  nnoooaatttmmmrrrrrrooow  ooo'
```

This result makes L more compressible, where L can be shrunk (reversibly) using methods such as *run-length encoding (RLE)*, where runs of repeated characters are replaced with a shorter code.

Effectiveness of the String Compression II

However in general, the computation of the sorting of the conjugates of a word is rather slow!

A more efficient way to implement algorithm A is to reduce the problem of sorting the rotations of the input string to that of sorting the suffixes of a similar string. We will use $T' =$ '*banana$*' as the input string with an *EOF* character to illustrate the implementation of BWT via the suffix array.

Let M be the matrix as defined in Algorithm A, whose rows consists of the rotations of T' sorted in a lexicographical order. Denote $SA_{T'}$ as the suffix array of T'.Then, we have

$$M = \begin{bmatrix} \$ & b & a & n & a & n & a \\ a & \$ & b & a & n & a & n \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ b & a & n & a & n & a & \$ \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \end{bmatrix}, SA_{T'} = \begin{bmatrix} 6 \\ 5 \\ 3 \\ 1 \\ 0 \\ 4 \\ 2 \end{bmatrix},$$

$$\text{Suffixes given by } SA_{T'} = \begin{bmatrix} \$ \\ a\$ \\ ana\$ \\ anana\$ \\ banana\$ \\ na\$ \\ nana\$ \end{bmatrix}$$

Definition 2.3.1

Let $L[i]$ denote the character at 0-based offset i for indexing in L, and let $SA_T[i]$ denote the suffix at 0-based offset i for indexing in L. Then for an input string T with the unique EOF character \$,

$$L[i] = \begin{cases} T[SA_T[i] - 1] & \textit{if} \quad SA_T[i] > 0 \\ \$ & \textit{if} \quad SA_T[i] = 0. \end{cases}$$

Example 2.3.2.

Let T = mississippi\$ be a string, where a \$ symbol is used to denote the end-of-string. Let L be the array that contains the final BWT output, given in the last column of the table below.

Suffixes	ID	Sorted Suffixes	Suffix Array	Sorted Rotations (A_s matrix)	BWT Output (L)
mississippi$	1	$	12	$mississippi	i
ississippi$	2	i$	11	i$mississipp	p
ssissippi$	3	ippi$	8	ippi$mississ	s
sissippi$	4	issippi$	5	issippi$miss	s
issippi$	5	ississippi$	2	ississippi$m	m
ssippi$	6	mississippi$	1	mississippi$	$
sippi$	7	pi$	10	pi$mississip	p
ippi$	8	ppi$	9	ppi$mississi	i
ppi$	9	sippi$	7	sippi$missis	s
pi$	10	sissippi$	4	sissippi$mis	s
i$	11	ssippi$	6	ssippi$missi	i
$	12	ssissippi$	3	ssissippi$mi	i

Now, for $T' =$ '*banana\$*', we rewrite T' with *T-ranking* to get $T' = b_0 a_0 n_0 a_1 n_1 a_2 \$$. Note that \$ is not ranked as it is unique. Then by Algorithm A, we get

$$M = \begin{bmatrix} \$ & b_0 & a_0 & n_0 & a_1 & n_1 & a_2 \\ a_2 & \$ & b_0 & a_0 & n_0 & a_1 & n_1 \\ a_1 & n_1 & a_2 & \$ & b_0 & a_0 & n_0 \\ a_0 & n_0 & a_1 & n_1 & a_2 & \$ & b_0 \\ b_0 & a_0 & n_0 & a_1 & n_1 & a_2 & \$ \\ n_1 & a_2 & \$ & b_0 & a_0 & n_0 & a_1 \\ n_0 & a_1 & n_1 & a_2 & \$ & b_0 & a_0 \end{bmatrix}, F = \begin{bmatrix} \$ \\ a_2 \\ a_1 \\ a_0 \\ b_0 \\ n_1 \\ n_0 \end{bmatrix} \text{ and } L = \begin{bmatrix} a_2 \\ n_1 \\ n_0 \\ b_0 \\ \$ \\ a_1 \\ a_0 \end{bmatrix}.$$

Next, by Algorithm B (Reverse Transform), similar to the example shown in the previous section, we have

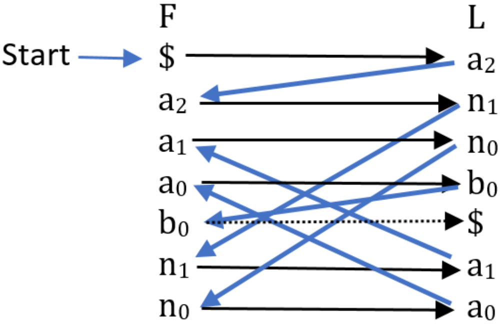


Figure: Reverse BWT starting at the right-hand side of T and moving left-wards

The FM Index (Full-text index in Minute space) of T is a space-efficient (compressed) full-text substring index of T, that is based on the Burrows-Wheeler transform (BWT), and bears similarity to the suffix array data structure.

In other words, the FM Index compresses the data and indexes it concurrently!

Definition 2.4.1. (B-Ranking)

Rank the characters in L according to the number of times the same character occurred previously in L.

By Algorithm A and definition 2.4.1, we update M to get

$$F = \begin{bmatrix} \$ \\ a_0 \\ a_1 \\ a_2 \\ b_0 \\ n_0 \\ n_1 \end{bmatrix}, L = \begin{bmatrix} a_0 \\ n_0 \\ n_1 \\ b_0 \\ \$ \\ a_1 \\ a_2 \end{bmatrix}, \textit{and Rank matrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 2 \end{bmatrix}.$$

Let P be a prefix of T. Suppose that we are searching for a string $P = ban$ in M (we continue with our results in the previous slide). We begin by searching for the rows of M that begins with the shortest proper suffix of P, given by n. In other words, these are rows that lie in the highlighted region:

$$\begin{bmatrix} F & & & & & & L & Rank \\ \$ & b & a & n & a & n & a & 0 \\ a & \$ & b & a & n & a & n & 0 \\ a & n & a & \$ & b & a & n & 1 \\ a & n & a & n & a & \$ & b & 0 \\ b & a & n & a & n & a & \$ & 0 \\ n & a & \$ & b & a & n & a & 1 \\ n & a & n & a & \$ & b & a & 2 \end{bmatrix}$$

Backwards Matching II

Next, we search for the rows that begins with the next-longest proper suffix of P, given by *an* (This can be done by the LF Mapping):

$$\begin{bmatrix} F & & & & & & L & Rank \\ \$ & b & a & n & a & n & a & 0 \\ a & \$ & b & a & n & a & n & 0 \\ a & n & a & \$ & b & a & n & 1 \\ a & n & a & n & a & \$ & b & 0 \\ b & a & n & a & n & a & \$ & 0 \\ n & a & \$ & b & a & n & a & 1 \\ n & a & n & a & \$ & b & a & 2 \end{bmatrix}$$

Finally, we search for the final suffix of P, which is *ban*. Similarly, we look at the characters that lie in the highlighted region in L, and observe that the occurrences of *an* are preceded by n_1 and b_0. However, since we want *ban* and not *nan*, this leads us to the final highlighted region:

$$\begin{bmatrix} F & & & & & & L & Rank \\ \$ & b & a & n & a & n & a & 0 \\ a & \$ & b & a & n & a & n & 0 \\ a & n & a & \$ & b & a & n & 1 \\ a & n & a & n & a & \$ & b & 0 \\ \boxed{b} & a & n & a & n & a & \$ & 0 \\ n & a & \$ & b & a & n & a & 1 \\ n & a & n & a & \$ & b & a & 2 \end{bmatrix}$$

Backwards Matching IV

Hence, for backwards matching, we apply LF Mapping over and over again to find the range of rows which are prefixed by increasingly longer proper suffixes of P, till the size of the range is equal to the number of times P occurs in T, or till the range becomes $\emptyset$, which corresponds to the case where we run out of suffixes or when P does not occur in T.

But.. searching for preceding characters in L is slow! In fact, it takes $O(n)$ time, where $n = |T|$. However, this can be made into $O(1)$ time by using a $n \times |\sum|$ *RanksChars* matrix.

At each row of *RankChars*, each entry is an integer that corresponds to the number of times the character has been observed up to and including the particular position in L. Continuing from the example with $T =$ '*banana\$*', we have:

$$\begin{bmatrix} F & L \\ \$ & a \\ a & n \\ a & n \\ a & b \\ b & \$ \\ n & a \\ n & a \end{bmatrix}, \qquad RanksChars = \begin{bmatrix} \$ & a & b & n \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 2 \\ 1 & 1 & 1 & 2 \\ 1 & 2 & 1 & 2 \\ 1 & 3 & 1 & 2 \end{bmatrix}.$$

Backwards Matching VII

Thus, by finding out the appropriate character c in *RanksChars* at the extreme ends of the range, we will be able to implement a method similar to backwards matching in $O(1)$ time.

In this case, should the character c occur more than once, the findings will return the ranks of the occurrences. Moreover, the character c does not occur when there is no difference between the two findings.

Now, we remove almost all the rows in the *RanksChars* matrix, and denote the rows kept behind as *rank offset*. So, every time we scan through $RankChars[c][i]$, we either find a row i that was not removed or a row i that was removed.

For the first case, we continue the scan till we find a row i that was removed, and for the second case, we scan the characters in L from i, and move our search forwards or backwards till we arrive at the next *rank offset*.

Backwards Matching IX

Recall that in computer science, an offset within the suffix array SA_T is an integer that indicates the distance between the beginning of the array and a given element i, within SA_T. Thus, to find out where P occurs in T (P's offset in T), we can simply look up SA_T.

Using our previous example where we were searching for a string $P = ban$, we arrive at:

$$\begin{bmatrix} F & & & & & & L & SA_T \\ \$ & b & a & n & a & n & a & 6 \\ a & \$ & b & a & n & a & n & 5 \\ a & n & a & \$ & b & a & n & 3 \\ a & n & a & n & a & \$ & b & 1 \\ \boxed{b} & \boxed{a} & \boxed{n} & \boxed{a} & \boxed{n} & \boxed{a} & \boxed{\$} & \boxed{0} \\ n & a & \$ & b & a & n & a & 4 \\ n & a & n & a & \$ & b & a & 2 \end{bmatrix}.$$

Backwards Matching XI

From SA_T, the match occurs at offset 0. However, to use less space than storing n integers in SA_T, we remove majority of the elements in SA_T, and generate them when required. Suppose that we store every 4^{th} entry of SA_T instead of every entry. When we look up $SA_T[1]$, we find that it has been removed ('-' in highlighted region below):

$$\begin{bmatrix} F & & & & & & L & SA_T \\ \$ & b & a & n & a & n & a & 6 \\ a & \$ & b & a & n & a & n & - \\ a & n & a & \$ & b & a & n & - \\ a & n & a & n & a & \$ & b & - \\ b & a & n & a & n & a & \$ & 0 \\ n & a & \$ & b & a & n & a & - \\ n & a & n & a & \$ & b & a & - \end{bmatrix}.$$

Backwards Matching XII

Then by the LF Mapping, we arrive at the next row:

$$\begin{bmatrix} F & & & & & & L & SA_T \\ \$ & b & a & n & a & n & a & 6 \\ a & \$ & b & a & n & a & n & - \\ a & n & a & \$ & b & a & n & - \\ a & n & a & n & a & \$ & b & - \\ b & a & n & a & n & a & \$ & 0 \\ n & a & \$ & b & a & n & a & - \\ n & a & n & a & \$ & b & a & - \end{bmatrix}.$$

However, we end up in a row that has been removed. So repeating the process, we eventually reach a retained row after 5 steps by the LF Mapping:

$$\begin{bmatrix} F & & & & & & L & SA_T \\ \$ & b & a & n & a & n & a & 6 \\ a & \$ & b & a & n & a & n & - \\ a & n & a & \$ & b & a & n & - \\ a & n & a & n & a & \$ & b & - \\ \boxed{b} & \boxed{a} & \boxed{n} & \boxed{a} & \boxed{n} & \boxed{a} & \boxed{\$} & \boxed{0} \\ n & a & \$ & b & a & n & a & - \\ n & a & n & a & \$ & b & a & - \end{bmatrix}.$$

Since at this row we have $SA_T = 0$ and 5 steps were taken to arrive at this row, the row we started the process has an offset $|5 - 0| = 5$.

Hence, searching for the offset of T corresponding to a row of M is $O(1)$, when retaining an element of SA_T at every k^{th} index of T.

In summary, the FM Index is a combination of L and an auxiliary data structure. This gives us the following definition for the FM Index[22]:

Definition 2.4.2.

Let $T[0..n-1]$ be a string of length $|T| = n$, and $SA_T[0..n-1]$ be its suffix array. The FM Index of T stores the following data structures:

1. The output string of BWT is defined as a string of characters $L[0..n-1]$, where

$$L[i] = \begin{cases} T[SA_T[i]-1] & \textit{if} \quad SA_T[i] \neq 0 \\ T[n-1] & \textit{if} \quad SA_T[i] = 0. \end{cases} \tag{1}$$

   So, L is an array of preceding characters of the sorted suffixes.

Definition 2.4.2. (Con't.)

2. For every $c \in \sum$, $C[c]$ is an array that stores the the total number of occurrences of characters that are lexicographically smaller than c. For example, for $T = banana\$$, we have $C[a] = 1, C[b] = 4, C[n] = 5, C[z] = 7$.
3. A data structure that supports $O(1)$ time computation of $occ(c, i)$, where $occ(c, i)$ is the number of occurrences of c in $L[0..i-1]$, for $c \in \sum$.

Outline of Tests

1. Apply BWT to space delimited data sets which comprises of binary text, and letter-based texts
2. For the letter-based texts, begin with tests on strings made up of characters from $\sum$, such that $|\sum| = 2$ (For binary texts, $\sum = \{0, 1\}$)
3. After transforming the data with the BWT, use the .ZIP archive file format to compress the data files
4. Proceed with tests on other texts made up of characters from $\sum$, where $|\sum| = n$ for increasing n

Remark

During all our tests, we first record 100 observations for each data file of a particular size in bytes, using a pseudo-random text generator to generate 100 random strings with the same file size (number of characters) in bytes. We will then proceed to apply the BWT to each randomly-generated string, and finally apply .ZIP to both *non-BWT* and *BWT* texts (strings).

Implementation of the Transform III

Definition

The formula for the *Compression Ratio* is given by

$$\text{Compression Ratio} = \frac{\text{Uncompressed Data Size}}{\text{Compressed Data Size}} \tag{2}$$

In general, a compression ratio <1 indicates that the size of the compressed file is greater than that of the original file, so compression will be in-favourable in this case.

Compression for Binary Data I

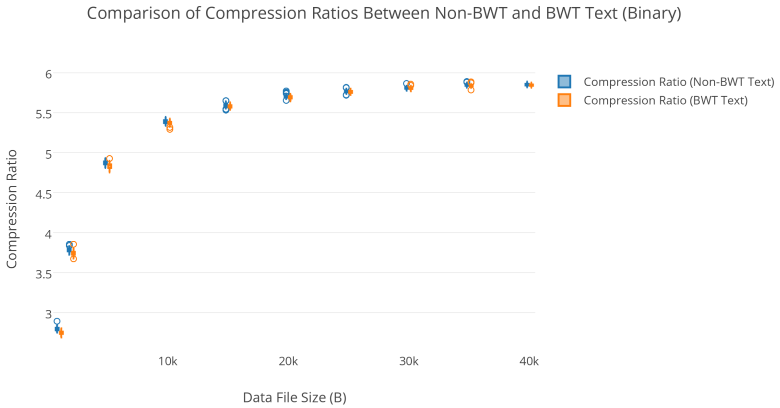


Figure: Test Results on binary ($\sum = \{0, 1\}$) strings

Compression for Binary Data II

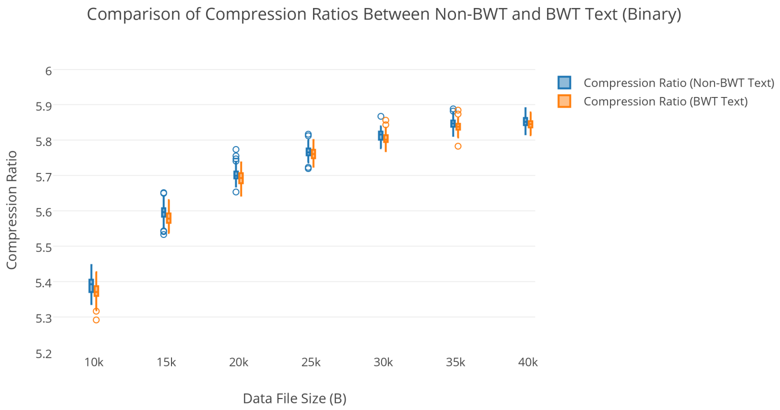


Figure: Zoomed-in portion of test Results on binary ($\sum = \{0, 1\}$) strings

Compression for Letter-Based Texts I

We begin our tests with $\sum = \{a, b\}$ for $|\sum| = 2$, followed by $\sum = \{a, b, c, d\}$, up till $\sum = \{a, b, \ldots, z\}$ for $|\sum| = 26$, with an increment of 2 characters for each test.

From our results, we observe that as the number of types of characters increases in a string, the lower the peak compression ratio r_i $(i \in \{x|x = |\sum|\})$ becomes, for each data file of a particular size.

Compression for Letter-Based Texts III

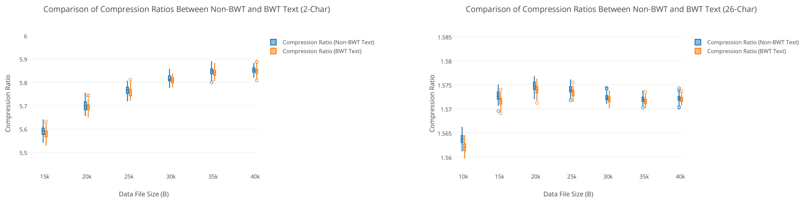


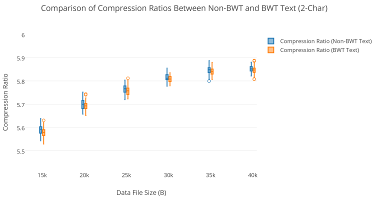
Figure: Zoomed-in portion of the histogram showing that BWT text have on average, lower compression ratios than non-BWT text for 2-character and 26-character strings

For example, the peak compression ratio r_2 for the test where $\sum = \{a, b\}$ is about 5.88, whereas the peak compression ratio r_{26} for the test where $\sum = \{a, b, \ldots, z\}$ is about 1.57.

Compression for Letter-Based Texts IV

More Findings

1. As $|\sum|$ increases, the compression ratio reaches its peak at lower file sizes
2. Both compression ratios for Binary ($\sum = \{0, 1\}$) and 2-Character ($\sum = \{a, b\}$)texts have peak r_i at about 35000B



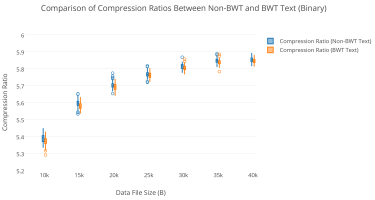
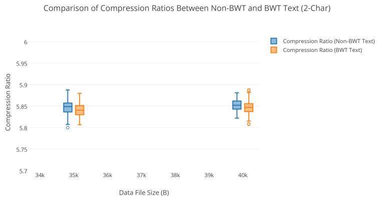


Figure: Zoomed-in portion of the histogram showing that BWT text have on average, lower compression ratios than non-BWT text for 2-character and Binary strings

However, the differences were marginal!

Compression for Letter-Based Texts VII



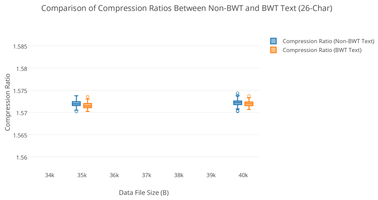


Figure: Zoomed-in portion of the histogram showing that BWT text have on average, lower peak compression ratios than non-BWT text for 2-character and 26-character strings

Summary of Findings

1. Difference in compression ratios are only significant for smaller file sizes, such as 1000B
2. Spread of compression ratios decreases as file size increases
3. Difference in mean compression ratios between non-BWT and BWT texts decreases as file size increases
4. The more random the data, the lower the effectiveness of BWT

The results of our testing can be found on the GitHub repository at: `https://github.com/weihao94/Burrows-Wheeler-Transformation-and-its-Applications`.

Definition 4.1.1.

A graph G is an ordered pair (V, E), where V is the set that comprises of vertices of G, and E is a set of ordered or unordered pairs of vertices u, v in G. G is said to be a directed graph if E is a set of ordered pairs of vertices (u, v), for some $u, v \in V$. G is said to be undirected if E is a set of unordered pairs of vertices $\{u, v\}$, for some $u, v \in V$.

Definition 4.1.2.

A **multigraph** G consists of a non-empty finite set $V(G)$ of vertices and a finite set $E(G)$ (possibly empty) of edges such that each edge joins two distinct vertices in $V(G)$, and any two distinct vertices in $V(G)$ are joined by a finite number (including zero) of edges.

Definition 4.1.3.

1. A $x - y$ **walk** is an alternating sequence $W: \ x = v_0 e_1 v_1 e_2 \dots v_{k-1} e_k v_k = y$ where $v_i \in V(G)$ for $i = 0, 1, \dots, k$, and $e_i \in E(G)$ for $i = 1, 2, \dots, k$ is an edge incident with v_{i-1} and v_i. The $x - y$ walk also has an initial vertex $x = v_0$ and terminal vertex $y = v_k$.
2. A $x - y$ **trail** is a $x - y$ walk where the edges in W are all distinct. In other words, every $x - y$ trail is a $x - y$ walk in G but a $x - y$ walk is a $x - y$ trail $\iff$ none of the edges in the walk are repeated.

Definition 4.1.3. (Con't.)

3. A $x - y$ **path** is a $x - y$ walk in which the vertices in W are all distinct. Thus, every $x - y$ path is a $x - y$ trail, but a $x - y$ trail is a $x - y$ path $\iff$ none of the vertices are repeated.
4. A $x - y$ walk is said to be **open** if $x \neq y$ and **closed** if $x = y$.
5. The **length** of the walk, trail or path is the umber of edges in W.
6. A closed trail of length at least two is called a **cycle** if $v_0, \ldots, v_{k-1}$ are all distinct.

Definition 4.1.4.

Let G be a connected multigraph. A trail in G is said to be an **Eulerian trail** of G if it contains all the edges of G. G is said to be **Eulerian** (resp. **semi-Eulerian**) if $\exists$ a closed (resp. open) Eulerian trail in G.

Theorem 4.1.5.

Let G be a connected multigraph. Then the following statements are equivalent:

1. G is Eulerian.
2. Every vertex of G is even.
3. The set $E(G)$ can be partitioned into cycles.

Corollary 4.1.6.

A connected multigraph G is semi-Eulerian $\iff$ G contains exactly two odd vertices. Furthermore, any open Eulerian trail in G must start at one of the odd vertices and terminate at the other odd vertex.

Definition 4.1.8.

A connected graph G of order $n \geq 3$ is **Hamiltonian** if it contains a spanning cycle. If G is a Hamiltonian graph, then any spanning cycle of G is called a **Hamiltonian cycle** of G.

Fleury's algorithm [10]:

Let G be an Eulerian multigraph. Proceed with the following steps:

1. Select an arbitrary vertex v_0 in G and set $W_0 := v_0, i := 0, G_i := G, E_i := \emptyset$.
2. Suppose that a trail $W_i = v_0 e_1 v_1 \dots e_i v_i$ has been constructed. Choose an edge e_{i+1} from $E(G) - E_i$ such that $e_{i+1} = v_i v_{i+1}$ for some vertex v_{i+1} and unless there is no other alternative, e_{i+1} is not a bridge of G_i.
3. Update $W_{i+1} := W_i e_{i+1} v_{i+1}, E_{i+1} := E_i \cup \{e_{i+1}\}$. Remove the edge e_{i+1} from G_i, along with any isolated vertices in G_i. If the resulting graph has no more edges, the algorithm ends. Otherwise, let the resulting graph be G_{i+1}, increase i by 1, and return to step 2.

So, Fleury's algorithm generates an Euler circuit that starts and ends at the same vertex!

De Bruijn Graphs I

Definition 4.2.1.

A k-bit string b is said to be obtained from a k-bit string $a = a_1 a_2 \ldots a_k$ by a left-shift operation if $b_i = a_{i+1}$, for $i = 1, 2, \ldots, k-1$, where b_k may be arbitrary. Then

1. A left shift $a_1 a_2 \ldots a_k \to b_1 b_2 \ldots b_k$ is a cyclic shift if $b_k = a_1$.
2. A left shift $a_1 a_2 \ldots a_k \to b_1 b_2 \ldots b_k$ is a de Bruijn shift if $b_k \neq a_1$.

De Bruijn Graphs II

Definition 4.2.2.

A **de Bruijn graph** of order k, denoted by $G(k)$, is a directed graph with 2^k vertices, each labelled with a unique k-bit string. Vertex v_i is joined to vertex v_j by an arc if bit string v_j is obtainable from bit string v_i by either a cyclic shift (rotation), or a de Bruijn shift.

Furthermore, each arc of $G(k)$ is a **cyclic shift arc** or a **de Bruijn arc**, according to the shift operation it represents. Each arc is labelled by the first bit of the vertex where it originates from, followed by the label of the vertex where it terminates.

Remark 4.2.3.

The above definition leads us to some properties of the de Bruijn graph:

1. Every de Bruijn graph is Eulerian and Hamiltonian.
2. Every de Bruijn graph is strongly connected.
3. Every vertex has in-degree 2 and out-degree 2. The first bit in the label on one of the vertices to which it points to is 0, and the first bit in the label on the other vertex is 1.

Definition 4.3.1.

A **de Bruijn sequence** $B(k, n)$ of order n on an alphabet $\sum$ of size k is a binary string of length k^n, where the last bit is said to be adjacent to the first bit, and every possible binary n-tuple occurs exactly once.

Two de Bruijn sequences are said to be identical if one can be obtained from the other by a cyclic permutation. In particular, every de Bruijn sequence corresponds to an Eulerian cycle on a de Bruijn graph.

Theorem 4.3.2. (de Bruijn's Theorem [16]).

For each positive integer n, there are $2^{2^{n-1}-n}$ de Bruijn sequences of order n.

Example 4.3.3.

By Theorem 4.3.2, there are 2 distinct de Bruijn sequences $B(2,3)$, given by 00010111 and 11101000.

To construct a de Bruijn sequence of order n, we use Fleury's algorithm to construct an Eulerian cycle of the de Bruijn graph with dimension $n-1$. Then, record the sequence of arc labels on the Eulerian cycle.

De Bruijn Sequences in the Inverse BWT IV

Example 4.3.4.

Suppose we want to construct a $B(2,4)$ de Bruijn sequence of order 4 with length $16(=2^4)$ from the de Bruijn graph of dimension 3. By Fleury's algorithm, we have a de Bruijn graph of dimension 3:

Example 4.3.4. (Con't).

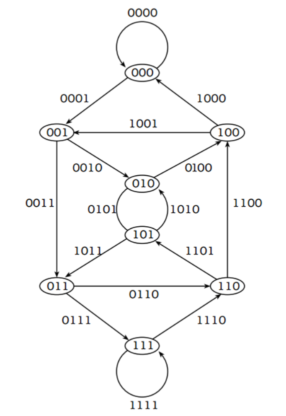


Figure: A de Bruijn graph of dimension 3 [1]

Example 4.3.4. (Con't).

Suppose we follow an Eulerian path through the nodes 000, 000, 001, 011, 111, 111, 110, 101, 011, 110, 100, 001, 010, 101, 010, 100, 000. From the output sequences, we get the de Bruijn sequence 0000111101100101 of length 16.

De Bruijn Sequences in the Inverse BWT VII

{0	0	0	0}	1	1	1	1	0	1	1	0	0	1	0	1
0	{0	0	0	1}	1	1	1	0	1	1	0	0	1	0	1
0	0	{0	0	1	1}	1	1	0	1	1	0	0	1	0	1
0	0	0	{0	1	1	1}	1	0	1	1	0	0	1	0	1
0	0	0	0	{1	1	1	1}	0	1	1	0	0	1	0	1
0	0	0	0	1	{1	1	1	0}	1	1	0	0	1	0	1
0	0	0	0	1	1	{1	1	0	1}	1	0	0	1	0	1
0	0	0	0	1	1	1	{1	0	1	1}	0	0	1	0	1
0	0	0	0	1	1	1	1	{0	1	1	0}	0	1	0	1
0	0	0	0	1	1	1	1	0	{1	1	0	0}	1	0	1
0	0	0	0	1	1	1	1	0	1	{1	0	0	1}	0	1
0	0	0	0	1	1	1	1	0	1	1	{0	0	1	0}	1
0	0	0	0	1	1	1	1	0	1	1	0	{0	1	0	1}
0}	0	0	0	1	1	1	1	0	1	1	0	0	{1	0	1
0	0}	0	0	1	1	1	1	0	1	1	0	0	1	{0	1
0	0	0}	0	1	1	1	1	0	1	1	0	0	1	0	{1

Figure: How the vertices of G s.t. $dim(G) = 3$ appear in the de Bruijn sequence

Definition 4.3.5.

A k-ary **necklace** of length n is an equivalence class under rotations of strings of length n over an alphabet $\sum$, where $|\sum| = k$. By BurnsidePolya enumeration, the number of k-ary necklaces of length n is

$$N_k(n) = \frac{1}{n} \sum_{d|n} \phi(d) k^{\frac{n}{d}} \tag{3}$$

where $\phi(n)$ is the number of integers in the interval $[1, n]$ that are relatively prime to n [1].

De Bruijn Sequences in the Inverse BWT - Lyndon Words II

Definition 4.3.6. [1,15].

A k-ary **Lyndon word** of length $n>0$ is a string of length n over an alphabet $\sum$, where $|\sum| = k$, and is the lexicographically smallest element in all its possible rotations. In other words, a Lyndon word corresponds to an aperiodic necklace representative.

Example

Let $\sum = \{0, 1\}$. Suppose we have a word 110010 over the alphabet $\sum$. The rotations of 110010 are: 110010, 100101, 001011, 010110, 101100, 011001. By listing these words in a lexicographical order, we obtain the Lyndon word 001011, which is the first element (and also the lexicographically smallest element) of the list: 001011, 010110, 011001 100101, 101100, 110010.

De Bruijn Sequences in the Inverse BWT - Lyndon Words III

Theorem 4.3.8. (Chen-Fox-Lyndon Theorem [11])

For every word w over an ordered alphabet $\sum$ that is non-empty, $\exists$ a unique factorization $w = v_t \dots v_1$ such that $v_1 \leq \dots \leq v_t$ is a non-decreasing sequence of Lyndon words.

Definition 4.3.9.

A permutation of a set S_n is a function $\pi : S_n = \{1, \dots, n\} \to S_n$ that is bijective.

Definition 4.3.10.

A permutation is a cyclic permutation $\iff$ it contains a single non-trivial cycle.

De Bruijn Sequences in the Inverse BWT - Lyndon Words IV

Algorithm C - De Bruijn Sequence by the Inverse BWT [7].

Suppose we have a string L made up of a size-k alphabet $\sum$ that is repeated k^{n-1} times, such that applying the Inverse BWT on L gives a string T that is of the same length of the de Bruijn sequence $B(k, n)$, and the result is a set of all Lyndon words of length d, where $d|n$, $k \geq 2$. To get a de Bruijn sequence $B(k, n)$, we proceed in the following manner:

De Bruijn Sequences in the Inverse BWT - Lyndon Words V

Algorithm C - De Bruijn Sequence by the Inverse BWT [7] (Con't).

1. Sort the characters in L, denote the output string as L'.
2. Place L' above L, and while preserving the order of the characters, map each character in L' to its corresponding position in L.
3. Write out the above permutation in a cycle notation, with the smallest position in each cycle first, and sort the cycles in ascending order.
4. In each cycle, replace every number with their corresponding letters in L', at that particular position.
5. Now, each cycle represents a Lyndon word sorted in a lexicographical order. Finally, we remove the parentheses to get the first de Bruijn sequence of $B(k, n)$.

De Bruijn Sequences in the Inverse BWT - Lyndon Words VI

Note that for every n and for every size-k alphabet $\sum$, there are $\frac{(k!)^{k^{n-1}}}{k^n}$ many distinct de Bruijn sequences $B(k, n)$.

Example 4.3.11.

Suppose for $n = 4, k = 2$, we want to create the first de Bruijn sequence $B(2,4)$ of length 2^4. By Algorithm C, we first concatenate the alphabet *ab* repeatedly for 8 times to get $L = abababababababab$ (of length 2^4). Then sort the characters in L, to get $L' = aaaaaaaabbbbbbbb$. Next, we place L' above L, numbering each column for the cycle notation, and map each character in L' to its corresponding position in L.

Example 4.3.11. (Con't).

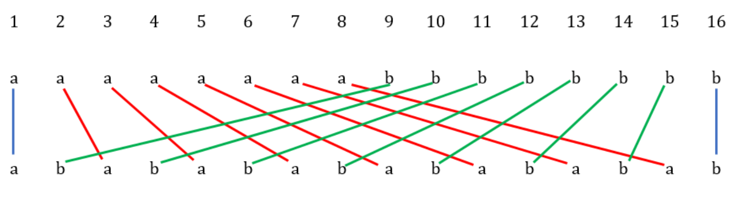


Figure: Illustration of the cycles of permutation by Algorithm C

Starting from the smallest number 1, the cycles are:

(1)(2 3 5 9)(4 7 13 10)(6 11)(8 15 14 12)(16).

De Bruijn Sequences in the Inverse BWT - Lyndon Words IX

Example 4.3.11. (Con't).

Next, replace each number in each cycle with the corresponding character in L' in each corresponding column to get (a)(aaab)(aabb)(ab)(abbb)(b). Note that these are Lyndon words of length d in lexicographical order, such that $d|4$ (for $n = 4$).

Finally, remove the parentheses to get $B(2,4) = aaaabaabbababbbb$, the first de Bruijn sequence of length $2^4 = 16$.

Bijective Variant of the BWT I

Overview: The bijective transform maps a string (or word) of length n to a string (or word) of length n without the need for any *EOF*-character or index.

Effectiveness: The bijective transform allows savings of several bits, and also strengthens data security during cryptographic operations.

Why is it used in place of the original BWT?

Bijective Variant of the BWT III

1. The *EOF*-character tends to speed up algorithms or simplify proofs, but it brings about new redundancies
2. $O(\log n)$ bits are needed to code the unique *EOF* character
3. It outperforms the BWT on nearly all the data files of the Calgary Corpus (a collection of text and binary data files - a benchmark for data compression in the 1990s) by at least a few hundred bytes
4. higher advantage than just preserving the rotational index

Definition 5.1.1. (Lyndon Factorization).

A word w can be factorized into factors such that each factor w_i is a Lyndon word (Recall from definition 4.3.6. that a k-ary Lyndon word of length $n>0$ is a string of length n over $\sum$ $s.t.$ $|\sum| = k$, and is the lexicographically smallest element in all its possible rotations).

Example 5.1.2.

Let w=$abacabab$. Then the Lyndon factorization of w gives us the factors $abac$, ab, ab.

Bijective Variant of the BWT V

Algorithm D - Bijective Transform.

Suppose we have an input string w of length n with Lyndon factorization $w = v_t \dots v_1$.

1. List out all possible rotations of each Lyndon word v_i.
2. Sort the list of rotated Lyndon words alphabetically by the first character.
3. Concatenate the last character of each rotated Lyndon Word to get the transformed word L.

Example 5.1.3.

Suppose we have a string $w = banana$. The Lyndon factorization is $w = v_4 \dots v_1$, where $v_4 = b, v_3 = an, v_2 = an$, and $v_1 = a$. In particular, $banana$ becomes $(b)(an)(an)(a)$, but the Lyndon words are combined into $(b)(anan)(a)$:

Index	**All Possible Rotations**
1	b
2	anan
3	nana
4	anan
5	nana
6	a

Example 5.1.3. (Con't).

Next, we sort the list of rotated Lyndon words alphabetically by their first character to get:

Index	All Possible Rotations
6	a
2	anan
4	anan
1	b
4	nana
5	nana

Hence, by concatenating the last character of each Lyndon word in the sorted list, we get $L = annbaa$, the output of the bijective transform.

Algorithm E - Inverse Bijective Transform.

Using L from Algorithm D, we proceed in the following steps (This is in fact largely similar to Algorithm C):

1. First sort the characters in L, and denote the resulting string as L'.
2. Place L' above L, and while preserving the order (index) of the characters, map each character in L' to its corresponding position in L.

Algorithm E - Inverse Bijective Transform. (Con't).

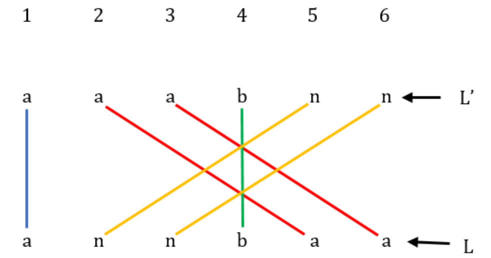
3. Write out the above permutation in a cycle notation, with the smallest position in each cycle first, and sort the cycles in ascending order. Alternatively, in place of Steps 1 and 2, one may derive the standard permutation π_L induced by L.
4. In each cycle, replace every number (index) with their corresponding letters in L', at that particular position (This yields a Lyndon factor v_i of L in each cycle).
5. Finally, by concatenating the cycles in a reverse-order (starting with cycles with the largest indexes), we obtain the original input string w.

Example 5.1.4.

In this example, we will use the string $w = banana$ from Example 5.1.3, and its output string $L = annbaa$ from the bijective transform to illustrate Algorithm E.

By Step 1 of Algorithm E, we have $L' = aaabnn$.

At Step 2, we obtain the following:



Example 5.1.4. (Con't).

Next, in Step 3, we obtain the cycles
$C_1 = (1), C_2 = (2,5), C_3 = (3,6), C_4 = (4)$.

Alternatively, one can derive the standard permutation π_L induced by L, given by

$$\pi_L = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 5 & 6 & 4 & 2 & 3 \end{pmatrix}$$

and then obtain the cycles $C_1, \ldots, C_4$ in a similar manner.

Example 5.1.4. (Con't).

Next, in Step 4, we replace every number in each cycle with their corresponding letters in L', at that particular position to get $C_1 = (a), C_2 = (an), C_3 = (an), C_4 = (b)$.

Finally, we concatenate the cycles in a reverse-order, starting with cycles with the largest index, and obtain the initial input string $w = banana$.

Conclusion & Summary[1]

[1] The slides can be found in my GitHub repository, together with the results of my tests at: `https://github.com/weihao94/Burrows-Wheeler-Transformation-and-its-Applications`

[6] Burrows, M., Wheeler, D.J.*A block sorting lossless data compression algorithm*. Digital Equipment Corporation, Tech. Rep. 124, 1994.

[7] Higgins, P.M. *Burrows-Wheeler transformations and de Bruijn words*. Theoretical Computer Science, Vol. 457, pp. 128-136, 2012.

[8] J. Gil, D. A. Scott. *A Bijective String Sorting Transform*. CoRR, abs/1201.3077, 2009.

Thank you for your kind attention! :)