Universiteit
Leiden
The Netherlands

Bachelor Thesis Informatica

The effect of alphabet reordering
on the bijective Burrows-Wheeler transform

Ruben Keurentjes

First supervisor:
Mark van den Bergh
Second supervisor:
Jonathan K. Vis

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

22/7/2026

Abstract

The Burrows-Wheeler transform (BWT) is a transformation that rearranges the characters of a string into runs of the same character. Compression algorithms are more effective when such runs exist and therefore the BWT is a useful step to improve the efficiency of lossless compression algorithms. However, the BWT is not bijective, which motivated the development of a bijective variant based on Lyndon factorization. This thesis investigates the concept of alphabet reordering and implements this concept for the bijective Burrows-Wheeler transform. Building on Wols' observation [Wol26], which reveals a new string property closely related to Lyndon words, this thesis also proves that this property cannot be used in the BBWT.

Contents

1 Introduction

In computer science, there is a never-ending quest to store more data in less space. This happens in the physical domain, where we try to fit more transistors in less space, but also in the digital domain, where we want more data to be stored in fewer bytes. An important technique for fitting more data in fewer bytes is compression, especially lossless compression. Lossless compression is a class of methods that try to compress files and are also able to decompress these files so that they contain their exact original information.
A common preprocessing step in text compression is the Burrows-Wheeler transform (BWT), or one of its variants. The BWT rearranges the characters of a string so that longer runs of the same character are created. One of its variants, the bijective Burrows-Wheeler transform (BBWT), makes use of Lyndon words, which are strings that are lexicographically smallest among their non-empty proper suffixes.
The BBWT often uses the ASCII table for its alphabet order in implementations. ASCII is a historically motivated order and may be suboptimal for compression. This thesis investigates changing the order used in the BBWT and determining whether better orders exist for this transform.
While researching synchronous games, Wols found interesting connections between the building blocks of the game Stack cherries and Lyndon words [Wol26]. This connection led to a variant of the Lyndon word, called an absolute Lyndon word. Since the BBWT uses Lyndon words, Wols suggests that it might be interesting to see whether a new version of the BBWT using absolute Lyndon words still works and if it results in better compressed results.

1.1 Related work

The BWT was first published by Burrows and Wheeler in 1994 [BW94] as part of a block-sorting approach to compression. A key concept of several BWT variants is that of *Lyndon words*. Lyndon words were first studied by Lyndon [Lyn54]. Lyndon words capture strings that are lexicographically minimal among their cyclic rotations, and they play an important role in factorization results for strings. The Lyndon factorization can be computed in linear time using Duval's algorithm [Duv83], which decomposes a string in a non-increasing sequence of Lyndon words.
Although the BWT is highly effective, it is not bijective without additional conventions. To address this problem, Gil et al. [GS09] proposed the BBWT. The BBWT provides a one-to-one mapping between input strings and transformed strings, removing ambiguity in inversion. This offers an alternative to the BWT, retaining the run-forming benefits and ensuring reversibility.
Albertini [AL23] highlights that Lyndon factorization is order-dependent: changing the underlying total order on the alphabet can change which substrings qualify as Lyndon words. This matters since many algorithms that build on Lyndon factorizations assume a fixed order. Altering this order can lead to different decompositions and potentially different behavior in algorithms using Lyndon factorization.
De Mol [dM21] introduced the concept of basic stackers, which was further studied by Wols [Wol26]. Wols proved that basic stackers have properties similar to Lyndon words, and therefore, Wols named these words *absolute Lyndon words*. Just like alphabet reordering, this might lead to different decompositions and thus potentially to different behavior in algorithms using Lyndon factorization.

1.2 Research question

The literature documents a range of applications of the BWT and BBWT. Although many variations are possible, the thesis by Wols [Wol26] conjectures that the incorporation of absolute Lyndon words may be a promising direction for further investigation. Thus, this thesis will investigate the effects of changing Lyndon words to absolute Lyndon words in the BBWT. The research question is therefore:

What is the effect of changing Lyndon words to absolute Lyndon words in the bijective Burrows-Wheeler transform with respect to compression performance?

Unfortunately, as proven in Section 2.5.3, due to the properties of absolute Lyndon words, we cannot answer this research question. As it turns out, the effect of changing Lyndon words to absolute Lyndon words in the BBWT results in a non-bijective transform. Therefore, we will not look into the use of absolute Lyndon words for a BWT-based variant.
Since absolute Lyndon words use a totally restructured order different from the canonical lexicographic order, we became interested in how a different alphabet order would affect the BBWT to construct the Lyndon factors and sort the rotations. The revised research question is:

What is the effect of alphabet reordering for the bijective Burrows-Wheeler transform with respect to compression performance?

We address the following subquestions to answer this research question:

1. What is the effect of the number of Lyndon factors of a string on the compression ratio of the BBWT using Nelson's reference implementation [Nel96]?

2. What alphabet order maximizes the compression ratio of the BBWT using Nelson's reference implementation?

To answer these questions, we will use three datasets from the Pizza & Chili Corpus [FN05] and one dataset from the Calgary Corpus [AB97]. All of these datasets contain strings, with different alphabet sizes, but we will map all of them to the ASCII table.

1.3 Thesis overview

In Section 1, we introduce our problem and research question with an overview of the thesis and related works. Section 2 explains the concepts and some basic algorithms for compression. Then it introduces definitions and algorithms for constructing the BWT and its variants. Section 2 also shows a deep dive into Lyndon words and absolute Lyndon words and ends with an overview of suffix arrays. Section 3 then describes our setup and describes the datasets that were used in the experiments. Then Section 4 describes the results of the experiments and analyzes those results. Finally, Section 5 concludes this thesis by answering the research question and discusses directions for future research.

2 Definitions

In this section, we look into the properties of the BWT, BBWT and the absolute BBWT. We describe their implementation and how they can be inverted. Concepts used by these transforms, such as Lyndon words, absolute Lyndon words, and suffix arrays, are also explained in this section.

2.1 Compression

Compression is the process of reducing the size of something. Although the object being compressed gets smaller, it still contains the same atoms.
A similar concept is studied in computer science. Datasets can be large and take up a lot of space on hard drives, and since data are fundamental to modern computing, we want to be able to store a lot of it. We can store more data by reducing its size, but we do not want to lose information in this data, since this makes it less valuable. Techniques to reduce the size of the data are called *compression*. As explained, the reduction of data in size is not useful if we lose information about this data. Therefore, this thesis looks into the concept of *lossless compression*, which are techniques that can reduce the size of data, and also have an inverse function that returns the data to its original size. This thesis studies the BWT, a data transformation to improve the effect of some lossless compression techniques. To understand this transform and how it affects these techniques, we must first understand some techniques that can benefit from this transform. These techniques can be used to describe the effect of the transform by showing the compressed size after some compression pipeline. The compression pipeline that we use is Nelson's reference implementation, which uses run-length encoding, some variant of the BWT, the move-to-front transform, and arithmetic coding. Therefore, we describe these algorithms in this section.

2.1.1 Run-length encoding

Run-length encoding (RLE) is a well-known algorithm in computer science, and arguably the most basic building block for compression algorithms. It is a lossless compression algorithm in which *runs* of data are stored as a single occurrence of the character and a count of its consecutive occurrences.

Definition 2.1. Let $x = x_1x_2\cdots x_n$ be a string over an alphabet Σ. A *run* in x is a maximal contiguous block of identical characters, i.e., there exist indices i and j with $1 \leq i \leq j \leq n$ such that

$$x_i = x_{i+1} = \cdots = x_j,$$

and the block cannot be extended:

$$(i = 1 \text{ or } x_{i-1} \neq x_i) \quad \text{and} \quad (j = n \text{ or } x_{j+1} \neq x_j).$$

The *run length* is $j - i + 1$, and the *run character* is x_i.

Example 2.2. Let $x = aaabccccdd\$$. The runs in x are

$$aaa,\ b,\ cccc,\ dd,\ \$.$$

Hence, the run-length encoding of x is the sequence of (character, length) pairs

$$(a, 3),\ (b, 1),\ (c, 4),\ (d, 2),\ (\$, 1).$$

Equivalently, we may write

$$\texttt{a}^3\ \texttt{b}^1\ \texttt{c}^4\ \texttt{d}^2\ \$^1,$$

where the exponent denotes the run length.

RLE is more efficient on data that contain longer runs. The run length is generally stored in binary. Otherwise, $22b$ can be decoded as two 2s and one b, but could also be decoded as 22 occurrences of the character b.

2.1.2 Move-to-front transform

Move-to-front (MTF) is a transform that replaces each character with its index in the list of recently used characters. After this, the character is moved to the front of this list. This results in long sequences of identical characters that are replaced by many zeros, whereas a character that has not been used in a long time is replaced by a larger number. The data is thus transformed into a sequence of integers, where if the data exhibits a lot of local correlations, then these integers are small.
In MTF, we assume that every character is a byte. Each byte value is encoded by its index in a list of bytes, and this list changes during the algorithm. Initially, the list is ordered by byte values, so each byte is represented by its own value. After encoding it, the value of this byte is moved to the front of the list. This process is repeated until the end of the string.

Example 2.3. Let $x = bananaaa$ and the list $(abcdefghijklmnopqrstuvw)$. The first character in the sequence is b, this character appears at index 1, and thus we encode b as 1. Then we move b to the front of the list $(bacdefghijklmnopqrstuvw)$. Now we want to encode the second character which is the character a. This character appears at index 1. Therefore, the encoding becomes $1, 1$, and the list $(abcdefghijklmnopqrstuvw)$. Continuing this process results in Table 2.1.

Table 2.1: result of iterating MTF over the string $bananaaa$.

Iteration	Sequence	List
1	1	$(abcdefghijklmnopqrstuvw)$
2	1,1	$(bacdefghijklmnopqrstuvw)$
3	1,1,13	$(abcdefghijklmnopqrstuvw)$
4	1,1,13,1	$(nabcdefghijklmopqrstuvw)$
5	1,1,13,1,1	$(anbcdefghijklmopqrstuvw)$
6	1,1,13,1,1,1	$(nabcdefghijklmopqrstuvw)$
7	1,1,13,1,1,1,0	$(anbcdefghijklmopqrstuvw)$
8	1,1,13,1,1,1,0,0	$(anbcdefghijklmopqrstuvw)$

Therefore, the final encoding is $1, 1, 13, 1, 1, 1, 0, 0$.

To decode the sequence back into the original string, start with the original list. Now decode the first value back to the original character and place the character to the front of the list. Repeat this process until the end of the string.

2.1.3 Arithmetic coding

Arithmetic coding is an entropy coding method that represents a string as a single number in the interval $[0, 1)$. Unlike Huffman encoding [Huf52], arithmetic coding does not assign a separate bit-string to each character. Instead, it refines an interval according to a probabilistic model and emits bits identifying a value inside the final interval [WNC87].

Definition 2.4. A model assigns a probability function $P(\cdot)$ over $\mathcal{A}$ with $\sum_{s\in\mathcal{A}} P(s) = 1$ and $P(s) > 0$. Define cumulative probabilities by

$$C(s_k) = \sum_{j=1}^{k-1} P(s_j),$$

so that each character s_k is associated with a sub-interval

$$I(s_k) = \big[C(s_k),\, C(s_k) + P(s_k)\big).$$

The encoder maintains an interval $[L, H)$, which is initialized as $[0, 1)$. For each character x_i, the interval is refined by selecting the sub-interval corresponding to x_i and mapping it into the range:

$$R = H - L, \qquad H' = L + R \cdot \big(C(x_i) + P(x_i)\big), \qquad L' = L + R \cdot C(x_i).$$

After processing all characters, any number in the final interval uniquely identifies the message [WNC87]. In practice, the encoder emits a finite bit sequence that selects a value guaranteed to lie inside the final interval.

Example 2.5. Let $\Sigma = \{a, b\}$ with $P(a) = 0.5$ and $P(b) = 0.5$, and let $x = aba$. Using the convention $C(a) = 0$ and $C(b) = 0.5$, start with $[L, H) = [0, 1)$.
Step 1:
$x_1 = a$. $R = 1$. New interval: $L = 0 + 1 \cdot 0 = 0$, $H = 0 + 1 \cdot 0.5 = 0.5$. So $[L, H) = [0, 0.5)$.
Step 2:
$x_2 = b$. $R = 0.5$. New interval: $L = 0 + 0.5 \cdot 0.5 = 0.25$, $H = 0 + 0.5 \cdot 1 = 0.5$. So $[L, H) = [0.25, 0.5)$.
Step 3:
$x_3 = a$. $R = 0.25$. New interval: $L = 0.25 + 0.25 \cdot 0 = 0.25$, $H = 0.25 + 0.25 \cdot 0.5 = 0.375$. Final interval: $[0.25, 0.375)$. Any value in this interval represents "aba" under the chosen model.

Decoding is performed by reversing the encoding process. The decoder maintains the same interval $[L, H)$ and uses the same model. Given a coded value $v \in [0, 1)$, it determines which character interval contains v, updates $[L, H)$ accordingly, and repeats. Termination is typically handled by encoding an explicit end-of-stream character or by transmitting the string length [WNC87].
A floating-point implementation is fragile because the interval may become arbitrarily small for longer strings. Therefore, arithmetic coders operate in an integer range $[0, T)$ with fixed precision. The encoder keeps the integer bounds `low` and `high` and updates them using integer arithmetic with cumulative frequencies rather than real-valued probabilities.
After each character, the encoder performs renormalization: whenever the current interval falls wholly in the lower half, wholly in the upper half, or in a middle "underflow" region, it outputs stable leading bits and rescales the interval, conceptually shifting left by one bit. This allows the encoder to emit bits incrementally while keeping `low` and `high` within fixed-precision integers, thereby avoiding loss of precision that would occur with naive real-number refinement [WNC87].

2.2 Suffix arrays

The BWT, BBWT and absolute Lyndon factorization heavily rely on suffix arrays (SA). Although for all of them, there are possibilities to write an algorithm without the use of SAs, these methods most often still rely on structures that use the same properties as SAs.

Definition 2.6. A suffix array of a string x of length n is an array $SA[0 \dots n-1]$ of integers that represent the starting positions of all suffixes of x sorted in lexicographical order. That is, if $SA[i] = k$ then the suffix $x[k \dots n-1]$ is the i-th smallest suffix of x in lexicographical order.

Example 2.7. Consider the string $x = banana\$$, then the suffixes in lexicographical order are:

$$\$, a\$, ana\$, anana\$, banana\$, na\$, nana\$$$

thus creating the suffix array:

$$[6, 5, 3, 1, 0, 4, 2]$$

Since the use of SAs is so important for all the algorithms in this thesis, it is necessary to use an implementation that uses an algorithm that creates a SA in linear time. Besides this we also require to make some changes in the algorithm to make it able to use our reordered alphabet. Therefore we need an algorithm that is simple to implement. Therefore, we the SA-IS algorithm [NZC09].

2.2.1 SA-IS

The first step for the SA-IS algorithm is to classify each character in a string as an S- or L-type.

Definition 2.8. [NZC09] Let $x \in \Sigma^*$ and let $y = x\$$ where $\$ \notin \Sigma$ is a unique sentinel with $\$ < c$ for all $c \in \Sigma$. Then $suf(y, i)$ is the suffix in y starting at $y[i]$.

Definition 2.9. [NZC09] Let $x \in \Sigma^*$ and let $y = x\$$ where $\$ \notin \Sigma$ is a unique sentinel with $\$ < c$ for all $c \in \Sigma$. A character $y[i]$ is S-type if

- it is the last character, or
- $y[i] < y[i+1]$, or
- $y[i] = y[i+1]$ and $suf(y, i+1)$ is S-type.

A character $y[i]$ is L-type if

- $y[i] > y[i+1]$, or
- $y[i] = y[i+1]$ and $suf(y, i+1)$ is L-type.

Example 2.10. The string $x = mmiissiissiippii\$$ has the following S- and L-types.

m	m	i	i	s	s	i	i	s	s	i	i	p	p	i	i	\$
L	L	S	S	L	L	S	S	L	L	S	S	L	L	L	L	S

The next step is to find the leftmost S-type and the LMS-substring.

Definition 2.11. [NZC09] A character $y[i], i \in [1, n-1]$, is called *leftmost S-type* (LMS) if $y[i]$ is S-type and $y[i-1]$ is L-type.

Definition 2.12. [NZC09] An LMS substring is

- a substring $y[i \dots j+1]$ for $i \neq j$, with both $y[i]$ and $y[j]$ being LMS characters, and there is no other LMS character in the substring; or
- the string containing only the sentinel.

Example 2.13. The string *mmiissiissiippii*\$ has the following LMS characters

m	m	i	i	s	s	i	i	s	s	i	i	p	p	i	i	$
		*				*				*						*

and has LMS-substrings *iissi*, *iissi*, *iippii*\$, \$

We now perform the induced sorting step. The main idea is to group suffixes by their first character and fill these groups (buckets) from the outside in.

Buckets. [NZC09] For each character c in the alphabet we create a bucket of size equal to the number of occurrences of c in the string. Conceptually, a bucket for c represents all suffixes whose first character is c. We order the buckets according to the alphabet order. We maintain two pointers per bucket:

- head$[c]$, the next free position on the left, and
- tail$[c]$, the next free position on the right.

Initialize with LMS suffixes. [NZC09] We initialize the suffix array $SA[0..n-1]$ with -1. For every LMS position p we place p into the tail of the bucket corresponding to the character $y[p]$, and decrement tail$[y[p]]$. Placing LMS positions from the right end of their buckets is important for the stability needed in the later right-to-left induction step.
After all LMS positions have been inserted, the bucket head pointers are reset to the leftmost positions of their buckets. The tail pointers will be reset again to the rightmost bucket positions before inducing S-type suffixes.

Insert L-type suffixes. [NZC09] Next, we scan the suffix array from left to right. Whenever we encounter a filled entry $SA[i] = p$, we consider the preceding position $q = p - 1$. If $q \geq 0$ and q is L-type, then the suffix starting at q must belong to the bucket of its first character $y[q]$. We therefore insert q at the current head$[y[q]]$ of that bucket and increment head$[y[q]]$.
Intuitively, each already placed suffix "pulls in" its immediate predecessor. Scanning left to right ensures that, within each bucket, L-type suffixes are induced in the correct relative (stable) order implied by the already sorted suffixes currently present in SA.

Insert S-type suffixes. [NZC09] After all L-type suffixes have been induced, the bucket tails are reset to the rightmost position of every bucket. The suffix array is then scanned from right to left. Whenever a filled entry $SA[i] = p$ is encountered, we consider the preceding position $q = p - 1$. If $q \geq 0$ and q is S-type, then the suffix starting at q belongs to the bucket of its first character $y[q]$. We insert q at the current tail of that bucket and decrement the corresponding tail pointer.
Scanning from right to left guarantees that the S-type suffixes are inserted in their correct relative order. Together with the previous left-to-right pass, this completes the induced sorting procedure.

Recursive reduction. [NZC09] After the first induced sorting, all LMS suffixes appear in lexicographical order. We then assign integer names to LMS substrings by comparing consecutive LMS substrings in that order. Two LMS substrings receive the same name if they are identical (character-by-character) up to and including the next LMS character (or the sentinel); otherwise, they receive different names.
Let the LMS positions in text order be $p_0 < p_1 < \cdots < p_{m-1}$. Using the assigned names, we form a reduced string y' of length m by setting

$$y'[t] = name(p_t) \quad for\ t = 0, \ldots, m-1.$$

If all names are distinct, then the lexicographical order of LMS suffixes is already fully determined and no recursion is needed. Otherwise, y' is strictly shorter than y, and SA-IS is applied recursively to y' to obtain the correct order of LMS suffixes, which is then mapped back to the corresponding LMS positions in y.

Final induced sort. [NZC09] Using the LMS order obtained either directly (unique names) or via recursion on y', we place the LMS positions into the tails of their corresponding buckets again. We then perform a final induced sorting pass: first inducing L-type suffixes by scanning left to right using bucket heads, and then inducing S-type suffixes by scanning right to left using bucket tails. This final induced sorting runs in $O(n)$ time, since each position is inserted at most once and bucket operations are constant-time given precomputed bucket boundaries. Combined with the fact that the recursive reduction strictly decreases the problem size, SA-IS runs in overall linear time.

Theorem 2.14. The SAIS algorithm is executed in $O(n)$ time.

Proof. As Nong, Zhang, and Chan [NZC09] proved, the time complexity for this algorithm is $O(n)$. We can explain this intuitively by looking at the algorithm step by step. Assigning each character a type from left to right can be executed in linear time. After this, we assign the LMS strings which can also be executed in linear time by looking at the types from the end of the string to the start of the string. Creating the buckets also takes at most linear time. The next step is inserting the L-type suffixes and S-type suffixes, which is done by going through all the buckets step by step and moving the heads and tails. Since we have n positions in all the buckets, we know that this also takes O(n) time. Then we calculate the shortened strings which reduces the problem by at least 50% each recursion. Since we know that

$$T(n) = T\left(\frac{n}{2}\right) + O(n) = O(n).$$

Therefore we conclude that the SAIS algorithm is executed in linear time. □

Example 2.15. [NZC09]
Let $y = mmiissiissiippii\$$. After calculating the LMS positions, we have the following table:

Index	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16
y	m	m	i	i	s	s	i	i	s	s	i	i	p	p	i	i	$
type	L	L	S	S	L	L	S	S	L	L	S	S	L	L	L	L	S
LMS			*				*				*						*

The next step is to initialize the buckets:

Bucket	$	i	m	p	s
SA	{16}	{-1, -1, -1, -1, -1, 10, 06, 02}	{-1, -1}	{-1, -1}	{-1, -1, -1, -1}

After inserting L-type suffixes we have filled the buckets as follows:

Step 2	$	i	m	p	s
After L-induce	{16}	{15, 14, -1, -1, -1, 10, 06, 02}	{01, 00}	{13, 12}	{09, 05, 08, 04}

After inserting S-type suffixes we have filled the buckets as follows:

Bucket	$	i	m	p	s
SA	{16}	{15, 14, 10, 06, 02, 11, 07, 03}	{01, 00}	{13, 12}	{09, 05, 08, 04}

Now we execute the recursive reduction step, where the names for the shortened string with suffixes $2, 6, 10, 16$ are 2,2,1,0. This eventually results in the suffix array

$$SA = [16, 15, 14, 10, 06, 02, 11, 07, 03, 01, 00, 13, 12, 09, 06, 08, 04].$$

2.3 Burrows-Wheeler transform

The Burrows-Wheeler transform (BWT) is a transform that rearranges the characters of a string into runs of the same characters. These runs are beneficial when used with compression techniques that rely on these runs, such as run-length encoding.

Definition 2.16. Let Σ be an alphabet and let $x \in \Sigma^*$ be a string of length $n = |x|$. For an integer k, the *k-fold cyclic rotation* of x is the string

$$\rho_k(x) \in \Sigma^n$$

defined by

$$(\rho_k(x))_i \;=\; x_{((i+k-1) \bmod n)+1} \qquad \text{for } i = 1, \ldots, n,$$

i.e.,

$$\rho_k(x) \;=\; x_{k+1}\, x_{k+2} \cdots x_n\, x_1 \cdots x_k,$$

where the indices are taken modulo n. In particular, $\rho_0(x) = x$ and $\rho_{k+n}(x) = \rho_k(x)$.

The transform is built by creating all the rotations of the given string, sorting all these rotations in lexicographical order, and then taking the last character of each rotation. Intuitively, a rotation means that we cut the string at some position and exchange the two parts: the first part moves to the end, while the last part moves to the front.

Definition 2.17. Let Σ be an alphabet. A *sentinel* is a special character \$ such that $\$ \notin \Sigma$. Given a string $x \in \Sigma^*$ we form $y = x\$$. We assume \$ is strictly smaller than every character in Σ, and \$ occurs exactly once (as the last character of y).

It is important that a unique end-of-string sentinel character is appended to every string. Otherwise, it will not be possible to reverse the BWT, which makes it unusable for lossless compression.
An algorithm for creating the BWT is given by Gil et al. [GS09] and is shown in Algorithm 2.3. Algorithm 2.3 combines Algorithm 2.1 and 2.2.

Example 2.18. Let $x = aababba$, then $\rho_4(x) = bbaaaba$.

Algorithm 2.1: CyclicRotations(x) [GS09]

Input: a string $x \in \Sigma^+$
Output: all rotations of the string x

```
1 n ← |x|
2 for i = 0 to n − 1 do
3 |   R ← R ∪ {ρ_i(x)}
4 return R
```

Theorem 2.19. [GS09] Algorithm 2.1, CyclicRotations(x), requires $O(n)$ time.

Proof. Algorithm 2.1 first takes the length of string x, which takes linear time to count. The for-loop in lines 2–3 is executed n times. Assuming that x is allocated in immutable storage, each of $\rho_i(x)$ can be represented by a triple of scalars: the address of the first character of x, the index i, and the length $n = |x|$ [GS09]. Therefore, line 3 is executed in constant time and line 2–3 is executed in linear time. Thus, we have a time complexity of

$$O(n) + O(n) \cdot O(1) = O(n).$$ □

Lexicographic order compares two strings from left to right and decides based on the first position where they differ. This happens similarly to comparing words in a dictionary using alphabetic order.

Algorithm 2.2: Last(R) [GS09]

Input: a set R

Output: the string composed of the last character of each of the members of R in lexicographical order

```
1 n ← |R|
2 η ← empty array of length n
3 for i ← 0 to n − 1 do
4 |   α ← min(R)                    // lexicographically smallest string in R
5 |   η[i] ← last(α)                                  // last character of α
6 |   R ← R \ {α}
7 return η
```

Definition 2.20. Let Σ be a finite alphabet equipped with a total order $\prec_{\text{lex}}$. The lexicographic order $\prec_{\text{lex}}$ on Σ^* is defined as follows: for $x, y \in \Sigma^*$, we write $x \prec_{\text{lex}} y$ iff either

i. there exists an index i such that
$x_j = y_j$ for all $j < i$ and $x_i \prec_{\text{lex}} y_i$, or

ii. x is a proper prefix of y.

Algorithm 2.2 sorts all the rotations and takes the last character of each string. One problem with this algorithm is that, as noted by Gil et al. [GS09], sorting the input set R takes $O(n \log n)$ comparisons in the worst case. Each comparison requires $O(n)$ character comparisons. Therefore, a naive implementation such as Algorithm 2.2 leads to a time complexity of $O(n^2 \log n)$.

Theorem 2.21. Let x be a string of length n on a totally ordered alphabet Σ, and let $x' = x\$$, where \$ is a unique sentinel character where $\$ \notin \Sigma$ and is smaller than all characters of Σ in lexicographical order. Let $R = \text{CyclicRotations}(x')$. Then Algorithm 2.2 can be implemented in $O(n)$ time.

Proof. Algorithm 2.2 sorts all the rotations of x' and outputs the last column. The naive implementation compares rotations character-by-character, but we can avoid this.
Since \$ occurs exactly once, any comparison between two rotations of x' encounters \$ before a full wrap-around. Therefore, the lexicographic order of all rotations of x' is exactly the lexicographic order induced by the corresponding suffixes of x'. Hence, instead of sorting all rotations directly, we compute the suffix array of x' [GRS07]. According to Theorem 2.14 a suffix array of a string of length $n + 1$ can be constructed in linear time. Once the suffix array is known, the BWT output character for each row is obtained in constant time, so producing the full last column takes linear time. □

Algorithm 2.3 combines Algorithm 2.1 and Algorithm 2.2 to compute the BWT. Given a string x, it first generates all rotations of x, then sorts these rotations lexicographically, and finally outputs the last character of each rotation in the sorted list. The purpose of the BWT is to reorder the input, which can then be compressed more effectively because this transform tends to create longer runs of equal characters.

Theorem 2.22. Algorithm 2.3 has linear time complexity.

Algorithm 2.3: Burrows–Wheeler Transform [GS09]

Input: a string α
Output: BWT(α)

```
1 R ← CyclicRotations(α)
2 B ← Last(R)
3 return B
```

Proof. Algorithm 2.1 has linear time complexity and Theorem 2.21 holds, thus the BWT can be formed in $O(n)$. □

As Theorem 2.21 shows, the BWT can be reduced to linear time, but no implementation has been presented. Therefore, we present an algorithm in Algorithm 2.4 that implements the BWT in linear time.

Algorithm 2.4: BWT using Suffix Array

Input: a string x of length n with sentinel \$
Output: the Burrows-Wheeler Transform BWT of x

```
1 SA ← suffix array of x
2 BWT ← empty array of length n
3 for i = 0 to n − 1 do
4 |   if SA[i] = 0 then
5 |   |   BWT[i] ← $
6 |   else
7 |   |   BWT[i] ← x[SA[i] − 1]
8 return BWT
```

Theorem 2.23. Algorithm 2.4 has linear time complexity.

Proof. Construction of the suffix array of a string x takes $O(n)$ time. The initialization of an array of length n also takes $O(n)$ time. The loop in lines 3–7 also executes exactly n times. In each iteration we perform a check of $SA[i] = 0$ in constant time, and one assignment to BWT$[i]$ in constant time. Therefore, the total running time is:

$$O(n) + O(n) + n \cdot O(1) = O(n).$$ □

Example 2.24. Let $x = banana\$$, then we have indices $x[0] = b, x[1] = a, x[2] = n, x[3] = a, x[4] = n, x[5] = a, x[6] = \$$. All suffixes sorted lexicographically are shown in Table 2.2. Therefore, we have the suffix array $SA = [6, 5, 3, 1, 0, 4, 2]$. Now we construct the BWT of x as shown in Table 2.3. This results in the transform $BWT(x) = annb\$aa$

2.3.1 Inverting the BWT

The BWT is a transform used in lossless compression, and therefore it has an inverse function that calculates the original input string. This is achieved by reconstructing the original input string

Table 2.2: The sorted suffixes of the string $x = banana\$$.

starting index of suffix	suffix
6	\$
5	$a\$$
3	$ana\$$
1	$anana\$$
0	$banana\$$
4	$na\$$
2	$nana\$$

Table 2.3: construction of BWT of the string $x = banana\$$.

index in BWT	$SA[i]$	suffix	$BWT[i]$
0	6	\$	$BWT[i] = x[SA[i] - 1] = a$
1	5	$a\$$	$BWT[i] = x[SA[i] - 1] = n$
2	3	$ana\$$	$BWT[i] = x[SA[i] - 1] = n$
3	1	$anana\$$	$BWT[i] = x[SA[i] - 1] = b$
4	0	$banana\$$	$BWT[i] = \$$
5	4	$na\$$	$BWT[i] = x[SA[i] - 1] = a$
6	2	$nana\$$	$BWT[i] = x[SA[i] - 1] = a$

from the transform. Reconstructing is performed using the last-to-first mapping (LF-mapping), which reconstructs the sorted matrix of all rotations of the input string.
The inverse is created over a totally ordered alphabet Σ, using a sentinel character $\$ \notin \Sigma$ such that $\$$ is lexicographically smaller than every character in Σ.

Definition 2.25. [BW94] Let Σ be a totally ordered alphabet with order $\prec_{\text{lex}}$. For $n \in \mathbb{N}$, define the sorting operator

$$R\colon \Sigma^n \to \Sigma^n$$

as follows: for any string $x \in \Sigma^n$, $R(x)$ is a unique string $y \in \Sigma^n$ such that

(i) y is a permutation of x, and

(ii) $y_1 \prec_{\text{lex}} y_2 \prec_{\text{lex}} \cdots \prec_{\text{lex}} y_n$.

Let $x \in \Sigma^*$ and let $L = BWT(x)$, where $n = |x|$ and $BWT(x) \in \Sigma^n$. We define

$$F = R(L).$$

We now define the LF-mapping.

Definition 2.26. Let Σ be an alphabet and let $x \in \Sigma^*$. Define $x[i \dots j]$ as the substring of x starting with the i^{th} character, and ending on the j^{th} character.

Definition 2.27. [BW94] Let Σ be a totally ordered alphabet. Let $x \in \Sigma^*$. Let $L = BWT(x)$. For each character $c \in \Sigma$, define $C[c]$ as the number of characters in L that are smaller than c:

$$C[c] = \#\{i | L[i] < c\}$$

and define $Occ(c, i)$ as the number of occurrences of c in the prefix $L[0 \dots i]$ for $i \in \{0, \dots, n-1\}$

$$Occ(c, i) = \#(\{i < n | L[i] = c\}).$$

The LF-mapping is the function $LF : \{0, \dots, n-1\} \times \Sigma^* \to \{0, \dots, n-1\}$ defined by

$$LF(i) = C[L[i]] + Occ(L[i], i) - 1.$$

Since the sentinel \$ occurs exactly once and is the smallest character, the row that starts with \$ is the first row in the sorted rotation matrix; equivalently, it corresponds to the index 0 in F (i.e. $F[0] = \$$). The inverse BWT reconstructs x by iterating LF.

Example 2.28. Let $L[0 \dots n-1] =$ `annb$aa` with $n = 7$ and in alphabetical order $\$ < a < b < n$.

Step 1: Compute the C array. We count how many characters in L are strictly smaller than each character:

$$\#(\$) = 1, \quad \#(a) = 3, \quad \#(b) = 1, \quad \#(n) = 2.$$

Hence

$$C[\$] = 0, \qquad C[a] = 1, \qquad C[b] = 1 + 3 = 4, \qquad C[n] = 1 + 3 + 1 = 5.$$

Step 2: Compute the Occ array. Define $Occ(c, 0) = 0$ and for $i = 0, \dots, n-1$: $Occ(L[i], i) = Occ(L[i], i-1) + 1$ (and other characters keep the same prefix count). For convenience, Table 2.4 lists the prefix counts for all characters.

Table 2.4: Prefix occurrence counts $Occ(c, i)$ for $L =$ `annb$aa`.

i	$L[i]$	$Occ(\$, i)$	$Occ(a, i)$	$Occ(b, i)$	$Occ(n, i)$
0	a	0	1	0	0
1	n	0	1	0	1
2	n	0	1	0	2
3	b	0	1	1	2
4	\$	1	1	1	2
5	a	1	2	1	2
6	a	1	3	1	2

Step 3: Reconstruct the string. First, find i such that $L[i] = \$$. Here $i = 4$. Then for $k = n-1, n-2, \dots, 0$ set $x[k] \leftarrow L[i]$ and update

$$i \leftarrow C[L[i]] + Occ(L[i], i).$$

The iterations are shown in Table 2.5.
Thus,

$$x[0..6] = \texttt{banana\$}.$$

After removing the end marker, we obtain the original string `banana`.

Table 2.5: Iterations of Algorithm 2.5 for $L = \texttt{annb\$aa}$.

k	current i	$x[k] = L[i]$	new $i = C[L[i]] + Occ(L[i], i)$
6	5	\$	$C[\$] + Occ(\$, 5) = 0 + 1 = 1$
5	1	a	$C[a] + Occ(a, 1) = 1 + 1 = 2$
4	2	n	$C[n] + Occ(n, 2) = 5 + 1 = 6$
3	6	a	$C[a] + Occ(a, 6) = 1 + 2 = 3$
2	3	n	$C[n] + Occ(n, 3) = 5 + 2 = 7$
1	7	a	$C[a] + Occ(a, 7) = 1 + 3 = 4$
0	4	b	$C[b] + Occ(b, 4) = 4 + 1 = 5$

Theorem 2.29. The inverse of the BWT can be constructed in linear time.

Proof. Let $L = BWT(x)$ be a string of length n over a totally ordered alphabet Σ, augmented with a unique sentinel \$ that is lexicographically smallest. First scan L once and compute the frequency that each character occurs. This takes $O(n)$ time. Next, compute for each character c, the number of characters in L that are strictly smaller than c. Since the order is known we can compute this array in $O(|\Sigma|)$ time. Define for each position i the rank

$$Occ(c, i) = \#(\{i < n | L[i] = c\}).$$

We can compute $Occ(c, i)$ in a single left-to-right pass by maintaining running counters seen$[c]$ and setting $Occ(L[i], i) \leftarrow \text{seen}[L[i]] + 1$ before incrementing seen$[L[i]]$. This takes $O(n)$ time. Let i be the index such that $L[i] = \$$, this is found in one scan, hence $O(n)$ in the worst case. For $k = n - 1, n - 2, \ldots, 0$ set

$$x[k] \leftarrow L[i], \qquad i \leftarrow C[L[i]] + Occ(L[i], i) - 1.$$

The loop executes exactly n iterations, and each iteration performs $O(1)$ array lookups and assignments, hence $O(n)$ total time. Since $|\Sigma|$ is constant, summing the costs yields

$$O(n) \ + \ O(|\Sigma|) \ + \ O(n) \ + \ O(n) \ = \ O(n). \qquad \square$$

2.4 Bijective Burrows-Wheeler transform

The Bijective Burrows-Wheeler transform (BBWT) is a reversible transform that reorders the characters so that equal characters form longer runs. The difference with the BWT is that the BBWT does not process the input as a single global block with a unique sentinel. Instead, it first decomposes the input string into a sequence of Lyndon words, using Chen–Fox–Lyndon factorization, and then applies the same transformation procedure to these factors [GS09].

2.4.1 Lyndon words

Lyndon words are the building blocks of the BBWT. Without Lyndon words, we cannot invert the BBWT back to the original string. In this section, we show that every string has a unique decomposition into Lyndon words.

Definition 2.30. [Lyn54] A string $x \in \Sigma^*$ is a Lyndon word if for every decomposition $x = uv$ with $u, v \in \Sigma^*$ we have $x \prec_{\text{lex}} vu$. Equivalently, x is strictly lexicographically smaller than each of its nontrivial rotations.

Example 2.31. The string *aaab* is a Lyndon word, since it is smaller than its rotations: *baaa*, *abaa*, *aaba*.

Lyndon words are, by Definition 2.30, cyclic, meaning that we have to generate every rotation of a string to know if it is a Lyndon word. On computers, this can be computationally expensive. To avoid this, Chen, Fox, and Lyndon also used a non-cyclic definition using suffixes.

Theorem 2.32. [CFL58] A string $x \in \Sigma^*$ is a Lyndon word if for every nonempty proper suffix s of x we have $x \prec_{\text{lex}} s$.

Proof. Let $x = uv$ and $u, v \in \Sigma^*$ and x be a Lyndon word. Then v is a proper nonempty suffix of x so according to the assumption $x \prec_{\text{lex}} v$. Since lexicographic order is preserved under right-concatenation, $x \prec_{\text{lex}} v$ implies that

$$x \prec_{\text{lex}} vu.$$

This is Definition 2.30. □

Example 2.33. The string *aaab* is a Lyndon word, since it is smaller than all its suffixes: *aab*, *ab*, and *b*.

Theorem 2.34. [CFL58] Let x be a string. There exists a unique decomposition $x = x_1 x_2 \dots x_n$ of x into Lyndon words $x_1, \dots x_n$, which is non-increasing in lexicographic order (i.e. $\succeq_{\text{lex}}$) $x_1 \succeq_{\text{lex}} x_2 \succeq_{\text{lex}} \dots \succeq_{\text{lex}} x_n$.

In addition to this, Duval proved three statements about the decomposition into Lyndon words.

Lemma 2.35. [Duv83] Let $x \in \Sigma^*$ and let its Lyndon decomposition be given by $x = x_1 x_2 \dots x_n$. Then the following properties hold:

(a) x_n is the lexicographically minimum suffix of x.

(b) x_n is the longest suffix of x that is a Lyndon word.

(c) x_1 is the longest prefix of x that is a Lyndon word.

Duval uses these statements to create an algorithm for Chen-Fox-Lyndon factorization that is executed in linear time. This algorithm is shown in Algorithm 2.5.

Theorem 2.36. Duval's algorithm (Algorithm 2.5) computes the Chen–Fox–Lyndon factorization of a string x in $O(n)$ time.

Proof. The initialization on lines 1–3 take constant time. The loop on lines 4–15 consist of a few steps. Observe that during executing the index j is monotonically increasing and is incremented only in line 12. Hence, over the entire algorithm, j can be incremented at most n times.
The index j starts a loop on $i + 1$. Then the first inner loop increases j by one for each step. The second loop adds a Lyndon factor to the array of factorizations and then increments i. When the second loop is finished the index i is incremented by exactly j. Therefore, we can conclude that Algorithm 2.5 is executed in linear time, since j is incremented at most n times. □

Algorithm 2.5: [CFL58] [Duv83]Chen-Fox-Lyndon Factorization (Duval's Algorithm)

Input: A string (or list) $x[0 \dots n-1]$
Output: A list of Lyndon factors

```
n ← |x|
factorization = []
i ← 0
while i < n do
    j ← i + 1
    k ← i
    while j < n and x[k] ≤ x[j] do
        if x[k] < x[j] then
            k ← i
        else
            k ← k + 1
        j ← j + 1
    // add new Lyndon words, if a Lyndon word occurs repeatedly, append each word separately
    while i ≤ k do
        factorization.append(x[i ... i + j − k − 1])   // add Lyndon word to factorization
        i ← i + j − k
```

2.4.2 Constructing the BBWT

After computing the Lyndon factorization over the input string, the BBWT generates all rotations of each Lyndon word, sorts these rotations, and then selects the last character of each word. An algorithm was proposed by Gil et al. [GS09] and is shown in Algorithm 2.6.

Algorithm 2.6: [GS09]Bijective Burrows-Wheeler Transform

Input: String x
Output: BBWT(x)

```
W ← Factor(x)                                   // Lyndon factorization of x
R ← ∅
for ω ∈ W do
    R ← R ∪ CyclicRotations(ω)
B = Sort(R)
C = Last(B)
return C
```

The algorithm proposed by Gil et al. has a time complexity of $O(n^2 \log n)$ due to the time complexity of Algorithm 2.2. Since we need an algorithm with a time complexity of $O(n)$, we need to alter this algorithm. Bannai et al. propose a better Algorithm [BKKP21]. This algorithm is very similar to

Algorithm 2.4, but does not use standard suffix arrays; instead, they use a modified suffix array variant that is aware of Lyndon factor boundaries. The algorithm proposed by Bannai et al. is shown in Algorithm 2.7.

Algorithm 2.7: [BKKP21]Bijective Burrows-Wheeler Transform using circular suffix array

Input: A string x of length n
Output: BBWT(x)

```
1  W ← Factor(x)                                          // Lyndon factorization of x
2  build arrays isStart[0..n − 1] and endAtStart[0..n − 1] from W
                                  // stores start and end index for each Lyndon factor
3  C ← CircularSuffixArray(x, W)
4  B ← empty array of length n
5  for i ← 0 to n − 1 do
6      p ← C[i]
7      if isStart[p] then
8          p ← endAtStart[p] − 1                          // wrap around inside the factor
9      else
10         p ← p − 1
11     B[i] ← x[p]
12 return B
```

Theorem 2.37. Algorithm 2.7 has linear time complexity.

Proof. **Preprocessing (lines 1–4):** From Lyndon factorization W we build two arrays: isStart[p] indicating whether position p is the start of a factor and endAtStart[p] storing the end position (exclusive) of the factor starting at p. This takes $O(n)$ time and $O(n)$ space.
Lines 5–10: The loop runs n times, and each iteration is $O(1)$:

- The check whether p is the start of a factor is one array lookup: isStart[p].
- If so, the wrap-around position is computed by one lookup and one subtraction: $p \leftarrow \text{endAtStart}[p] - 1$.
- Otherwise, $p \leftarrow p - 1$ is constant time.
- Finally, assigning $B[i] \leftarrow x[p]$ is constant time.

Hence, the loop requires $O(n)$ in total. □

2.4.3 Inverting the BBWT

The BBWT is used in lossless compression algorithms. Therefore, we can define the inverse of the transform. This is done by reconstructing the Lyndon words from the transform. This happens by constructing the LF-mapping, which reconstructs the sorted list of all rotations of the Lyndon words from the original input string, which is similar to the inverse of the BWT. Then we try to find the

cycles in this LF-mapping, from which we reconstruct the input string. Afterwards, we reconstruct the correct rotation of the Lyndon word by finding its smallest rotation. Then we sort the obtained Lyndon factors, since the Chen-Fox-Lyndon factorization returns a factorization of Lyndon factors in non-increasing lexicographic ordering. First, we need to redefine the Definition 2.25 to make sure that L uses the BBWT.

Definition 2.38. [GS12] Assume a totally ordered alphabet Σ. Let $x \in \Sigma^*$ be an input string. Let $L = BBWT(x)$ and $n = |L|$. $L \in \Sigma^n$ is the BBWT output of a string x. $F \in \Sigma^n$ is the string obtained by sorting the characters of L in lexicographic order.

We now create the LF-mapping.
Since LF is a permutation on the index set $\{0, \ldots, n-1\}$, it can be written as a set of disjoint cycles.

Definition 2.39. [GS12] Let p be a permutation of $\{0, \ldots, n-1\}$. A *cycle* is a sequence $(i_0, i_1, \ldots, i_{m-1})$ of distinct indices such that

$$p(i_t) = i_{t+1} \quad (0 \leq t < m-1), \qquad p(i_{m-1}) = i_0.$$

The *cycle decomposition* of p is the collection of its disjoint cycles.

Definition 2.40. [GS09] Let $(i_0, \ldots, i_{m-1})$ be a cycle of LF, and $i_0 = \mathrm{LF}(i_{m-1})$. Given a cycle $(i_0, \ldots, i_{m-1})$ of LF, define

$$\mathrm{Spell}(i_0, \ldots, i_{m-1}) = L[i_{m-1}], L[i_{m-2}] \cdots L[i_0].$$

So we traverse the cycle by repeatedly applying LF, read the characters from L at the visited indices, and reverse the obtained sequence. The resulting string has length m and is a rotation of the original factor that generated this cycle.
A cycle determines its underlying factor only up to cyclic rotation. We recover the unique Lyndon factorization by taking the lexicographically smallest rotation of each factor.
For each cycle γ of LF, compute $u_\gamma := \mathrm{Spell}(\gamma)$ and rotate the factor until it is the smallest lexicographically among all rotations. Finally, sort the recovered factors in non-increasing lexicographic order and concatenate them; this yields the inverse BBWT output.

Example 2.41. Let $L = BBWT(\texttt{banana}) = \texttt{annbaa}$ we obtain $C[a] = 0, C[b] = 3, C[n] = 4$ and $LF = [0, 4, 5, 3, 1, 2]$. This results in the cycle decomposition $P = \{(0), (1, 4), (2, 5), (3)\}$. Applying Definition 2.40 gives

$$\mathrm{Spell}(0) = \texttt{a}, \quad \mathrm{Spell}(1, 4) = \texttt{an}, \quad \mathrm{Spell}(2, 5) = \texttt{an}, \quad \mathrm{Spell}(3) = \texttt{b}.$$

The minimal rotations are the same strings, hence the recovered multiset of factors is $\texttt{a}, \texttt{an}, \texttt{an}, \texttt{b}$. Sorting these factors in non-increasing lexicographic order yields $\texttt{b}, \texttt{an}, \texttt{an}, \texttt{a}$, and concatenation returns `banana`.

2.4.4 BBWT using alphabet reordering

Theorem 2.32 states that a Lyndon word is lexicographically the smallest among its suffixes. Albertini and Louza [AL23] suggest that Lyndon factorization, without alphabet reordering, may not be an effective way to break strings into manageable pieces. This suggests that alphabet reordering could potentially lead to even more efficient ways of working with larger strings. They show the difference in number of factors and length of the longest factor for different alphabet orders. Since this leads to a different factorization, it might be interesting to investigate the effects of these different orders on the BBWT.
First, we need to create a new order for an alphabet Σ.

Definition 2.42. [AL23] Let $\Sigma = \{\alpha_1, \ldots, \alpha_m\}$ be a totally ordered alphabet with $\alpha_1 < \alpha_2 < \cdots < \alpha_m$. Let $\pi : \Sigma \to \Sigma$ be a permutation. We define a new total order $<_\pi$ on Σ by

$$a <_\pi b \quad \iff \quad \pi(a) < \pi(b).$$

Equivalently, the sequence $\pi^{-1}(\alpha_1), \pi^{-1}(\alpha_2), \ldots, \pi^{-1}(\alpha_m)$ is in increasing order under $<_\pi$.

Consequently, a string in Σ^* may have different orders when considering it over the lexicographic order and the permuted order $<_\pi$.

Example 2.43. Consider two strings $u = ban, v = ana$. We have $v < u$ when $u, v \in \Sigma^*$ using the lexicographic order, while $v > u$ when $u, v \in \Sigma^*$ with the permuted order, where $b <_\pi a <_\pi n$.

This might result in two strings having different Lyndon factors when factorized over different alphabets.

Example 2.44. Let $x = banana$ with $\Sigma = \{a, b, n\}$. Then the Lyndon factorization under regular order is $[b, an, an, a]$. When we factorize x over alphabet Σ with order $b <_\pi a <_\pi n$, then we obtain the Lyndon factorization $[banana]$.

As Albertini and Louza [AL23] notice, for an alphabet Σ with n characters, the number of possible orders is $n!$. This is small for alphabets such as DNA, where the possible orders are $4! = 24$. For larger alphabets such as ASCII the number of possible orders becomes too large to work with.

Albertini and Louza [AL23] also present two orders that could show interesting results in this research. First, they introduce the Most Frequent Symbol order (MFS). In the MFS, the most frequent character in a file is put in the first position in the order, followed by the second most frequent character is put in the second position, etc.

Example 2.45. Let $\Sigma = \{a, b, n\}$ and let $x \in \Sigma^*$. Consider the string $x = banana$, then the MFS for this string is $a <_\pi n <_\pi b$.

The other order that they introduce is the Least Frequent Symbol order (LFS). In the LFS, the least frequent character in a file is put in the first position, after which the second least frequent character is put in the second position, and so on in ascending order of frequency.

Example 2.46. Let $\Sigma = \{a, b, n\}$ and let $x \in \Sigma^*$. Consider the string $x = banana$ then the LFS for this string is $b <_\pi n <_\pi a$.

2.5 Absolute Lyndon factorization

In this section, we explain what absolute Lyndon words are. First, we explain what they are and how they can be used. We show that absolute Lyndon words, while using the signed-lexicographic order instead of the canonical lexicographic order, are closely related to Lyndon words. After this we show a way of factorizing a string in a unique factorization of absolute Lyndon words. In the last part, we prove that this factorization cannot be used for the BBWT.

The signed lexicographic order has for every character $\sigma \in \Sigma$ a character that we call the *negative* of σ, denoted by $\overline{\sigma} \in \Sigma$. You can take the negative of a character twice, resulting in the original character $\overline{(\overline{\sigma})} = \sigma$.

Definition 2.47. [Wol26] Let $\Sigma = \{\sigma_1, \sigma_2\}$ be an alphabet of two characters. We define the signed-lexicographic order $\prec_{\text{slex}}$ on strings $x \in \Sigma^*$ as follows. We look at the leftmost character where the two strings differ and then decide which string is smaller based on the order $\sigma_1 < \varepsilon < \sigma_2$ (assuming that $\sigma_1 < \sigma_2$).

Each string or substring has a weight, which is important for factorization.

Definition 2.48. [Wol26] Let Σ be the alphabet of two characters $\Sigma = \{\sigma_1, \sigma_2\}$, with $\sigma_1 < \varepsilon < \sigma_2$. Define the integer coding $\varphi\colon \Sigma \to \mathbb{Z}$ by

$$\varphi(\alpha) = \begin{cases} -1 & \text{if } \alpha = \sigma_1, \\ 1 & \text{if } \alpha = \sigma_2. \end{cases}$$

For a string $x = x_1 \cdots x_n \in \Sigma^*$, the *signed lexicographical weight* is

$$w(x) = \sum_{i=1}^{n} \varphi(x_i)\, 2^{-i}.$$

Example 2.49. For the string $x = aabab$ and alphabet $\Sigma = \{a, b\}$ and order $a < \varepsilon < b$ we get the signed lexicographical weight:

$$w(x) = \frac{1}{2} \cdot (-1) + \frac{1}{4} \cdot (-1) + \frac{1}{8} \cdot 1 + \frac{1}{16} \cdot (-1) + \frac{1}{32} \cdot 1 = -\frac{21}{32}$$

Definition 2.50. [Wol26] We define the *negative* $-x$ of a string x as the string where all characters are replaced by their negatives.

Definition 2.51. [Wol26] We define the *absolute value* of a string x as x if x starts with a positive character and $-x$ if x starts with a negative character:

$$abs(x) = \begin{cases} x & \text{if } x \geq \varepsilon, \\ -x & \text{if } x < \varepsilon. \end{cases}$$

We can combine all the given definitions in this subsection to get the definition of an absolute Lyndon word.

Definition 2.52. [Wol26] A string $x \in \Sigma^*$ is called an absolute Lyndon word if its absolute value is minimal among the absolute values of its suffixes, according to the signed lexicographical ordering.

The following definition tells us that we can split any word in two, by splitting off the weakest suffix x_i. It is important to notice that the weakest suffix x_i is a Lyndon word because the word is weaker than all of its suffixes.

Definition 2.53. [dM21] Let x be a string of length n and let $0 \leq i < n$ such that $abs(x[i \dots n]) \leq abs(x[j \dots n])$ for all $0 \leq j < n$. Then $x = x[1 \dots i-1] + x[i \dots n]$. Here, $x[1 \dots i]$ is the prefix of x consisting of the first i characters.

Notice that the suffix $x[i \dots n]$ automatically is an absolute Lyndon word, since it is weaker then all of its suffixes. Therefore, we can repeatedly split of absolute Lyndon words from a string. This way we can decompose any string into absolute Lyndon words [Wol26].

Example 2.54. Let $x = bbaababb$ and alphabet $\Sigma = \{a, b\}$ and order $a < \varepsilon < b$. Then we have the absolute Lyndon factorization $D = [b, baab, abb]$.

Now we can define the absolute Lyndon factorization as defined by Wols [Wol26].

Definition 2.55. [Wol26] Let x be a binary string. There exists a unique decomposition of x into absolute Lyndon words $x_1, \dots x_n$, with $|x_1| \succ_{\text{slex}} |x_2| \succ_{\text{slex}} \dots \succ_{\text{slex}} |x_n|$, and $|x_i| = |x_{i+1}|$ only if x_i and x_{i+1} start with the same character.

Since the absolute Lyndon factorization will replace the Chen-Fox-Lyndon factorization, we want the algorithm for the absolute version to run in linear time. If the absolute factorization can be executed in linear time, this enables a fair comparison of both factorizations with each other.
We will now look at two different ways to achieve absolute Lyndon factorization.

2.5.1 Absolute Lyndon factorization (signed lexicographical weight)

This factorization for absolute Lyndon words was proposed by Wols [Wol26]. It builds on the definition of signed lexicographic weight (Defenition 2.48). The first thing the algorithm does is to calculate the weight of each suffix of a string x beginning at the end of the string, then going to the front.

Theorem 2.56. To calculate the signed lexicographical weight of a suffix of x at position i we can use the following formula:

$$w(x[i \dots n]) = \begin{cases} \varphi(x_i) \cdot 2^{-1} + w(x[i+1 \dots n]) \cdot 2^{-1} & \text{if } i < n, \\ \varphi(x_i) \cdot 2^{-1} & \text{if } i = n. \end{cases}$$

where $\varphi(x_i)$ is the weight of the character at position i.

Proof. Using Definition 2.48, we write

$$\begin{aligned} w(x[i \dots n]) &= \sum_{j=i}^{n} \varphi(x_j) 2^{-(j-i+1)} \\ &= \varphi(x_i) 2^{-1} + \sum_{j=i+1}^{n} \varphi(x_j) 2^{-(j-i+1)}. \end{aligned}$$

Furthermore, we know that

$$\begin{aligned}\sum_{j=i+1}^{n} \varphi(x_j)2^{-(j-i+1)} &= 2^{-1} \sum_{j=i+1}^{n} \varphi(x_j)2^{-(j-(i+1)+1)} \\ &= 2^{-1} \cdot w(x[i+1 \dots n]),\end{aligned}$$

and therefore:

$$w(x[i \dots n]) = \varphi(x_i)2^{-1} + w(x[i+1 \dots n]) \cdot 2^{-1}. \qquad \square$$

It is obvious that this can be implemented in a function that calculates the signed lexicographical weight of each suffix of a string in linear time.
According to Definition 2.53, we can split a string into two words where the second is an absolute Lyndon word. This process can be repeated over and over for the second part of the string until the string is completely factorized into absolute Lyndon words. We will use this definition for our algorithm.

The algorithm will work following the next steps:

1. Start from the end going to the front of the string and calculate the weight of every suffix.
2. Start from the front of the string and look at the absolute value of the signed lexicographical weight of the word x. Recall the absolute weight of x and proceed until you find a suffix y with a smaller absolute weight than x. We can now cut the suffix y and get the first absolute Lyndon word. Then we repeat this for the suffix y until we cannot find a smaller absolute value.

Example 2.57. We want to split up the string $x = bbaababb$ of the alphabet $\Sigma = a, b$ and $a < \varepsilon < b$. b has weight 1 and a has weight -1.

Calculate every absolute value of each suffix using Theorem 2.56. The suffix b has weight $1 \cdot 2^{-1} = \frac{1}{2}$. The next suffix bb has weight $1 \cdot 2^{-1} + w(b) \cdot 2^{-1} = \frac{1}{2} + \frac{1}{4} = \frac{3}{4}$, suffix abb has weight $-1 \cdot 2^{-1} + w(bb) \cdot 2^{-1} = -\frac{1}{2} + \frac{3}{8} = -\frac{1}{8}$ etc.

This results in the absolute signed lexicographical weights of the string x in Table 2.6.

Table 2.6: The absolute signed lexicographical weights of all suffixes of string $x = bbaababb$.

b	b	a	a	b	a	b	b
$\frac{151}{256}$	$\frac{46}{256}$	$\frac{164}{256}$	$\frac{72}{256}$	$\frac{112}{256}$	$\frac{32}{256}$	$\frac{192}{256}$	$\frac{128}{256}$

Now, we start at the front and look at the weights of each suffix. The first suffix of x is $bbaababb$ and has weight $\frac{151}{256}$. Now we move forward until we find a smaller weight. We find this at the suffix $baababb$ with weight $\frac{46}{256}$. We cut the string before the character with this weight, and therefore split the string into two parts: $b, baababb$. Again, we move forward until we find a smaller weight, in this case smaller than $\frac{46}{256}$. We find this at suffix abb with weight $\frac{32}{256}$. If we move to the end of the string we cannot find a smaller weight than $\frac{32}{256}$ and therefore we now know that the factorization into absolute Lyndon words of the string $x = bbaababb$ is b, $baab$, abb.

This results in Algorithm 2.8.

Algorithm 2.8: Absolute Lyndon factorization

Input: A string $\alpha = \alpha_1\alpha_2 \dots \alpha_n$

Output: A list of absolute Lyndon factors

```
// Step 1: compute suffix weights
w[n] ← x_n · 2^{-1}
for i ← n − 1 to 1 do
    w[i] ← x_i · 2^{-1} + w[i + 1] · 2^{-1}

// Step 2: factorization
i ← 1
ℓ ← 1
k ← 1
while i ≤ n do
    j ← i
    min ← |w[i]|
    while j ≤ n and ¬foundSmaller do
        if |w[j]| < min then
            min ← |w[j]|
            factors[ℓ] ← α[i . . . j]
            ℓ ← ℓ + 1
        j ← j + 1
    k ← i
    i ← j
factors[ℓ] ← α[k . . . n]
return factors
```

Unfortunately, this algorithm leads to a problem that we need to address. This implementation uses a fraction in which the denominator is getting exponentially larger. This is not a problem on paper, but we assume a fixed-width machine-integer type of w bits. Hence, all intermediate values must fit in w bits, otherwise overflow occurs. We run into a storage problem in which we are not able to compute the weight precisely enough when we are working with strings of more than w characters, and that is, when we are working with just two characters in our alphabet. If we use more than two characters (which is not proven to work for absolute Lyndon words), such as the standard Latin alphabet, which contains 26 characters, we lose our precision after just $\frac{w}{\log_2 26}$ characters.

Let us assume we have two characters in our alphabet, then two sub-strings will have identical absolute weights in our memory if the first 64 characters of both substrings are identical. In addition to that, two substrings also have identical absolute weights if string a has a prefix s with 64 characters and string b has a prefix t with 64 characters with $abs(s) = t$.

Example 2.58. Suppose that we have alphabet $\Sigma = \{a, b\}$ and $a < \varepsilon < b$. The character b has weight 1 and the character a has weight -1. Then the absolute lexicographical weight of string $x = a$ is $abs(-\frac{1}{2})$ which is equal to the signed absolute lexicographical weight of string $y = b$, of which the absolute weight is $abs(\frac{1}{2})$.

Besides this, we also have the problem that it is possible in a 64-bit computer that two suffixes have equal weight if their indexes are next to each other.

Definition 2.59. We say that two consecutive suffixes $(x_k \dots x_n)$ and $(x_{k+1} \dots x_n)$ have equal *signed lexicographical weight* if and only if

$$abs(x \cdot 2^{-1} \pm 2^{-1}) = abs(x).$$

In which we add 2^{-1} if the first character x_n has a positive weight and subtract 2^{-1} if the first character x_n has a negative weight.

Example 2.60. Suppose that we have alphabet $\Sigma = \{a, b\}$ and $a < \varepsilon < b$. The character b has weight 1 and a has weight -1. For the following two characters to have equal absolute signed lexicographical weight, we would need to have that:

$$abs(\frac{x}{2} \pm \frac{1}{2}) = abs(x).$$

If the first character is an a, then

$$abs(\frac{x}{2} - \frac{1}{2}) = abs(x),$$

resulting in

$$-(\frac{x}{2} - \frac{1}{2}) = x \vee \frac{x}{2} - \frac{1}{2} = x.$$

If the first character is a b, then

$$abs(\frac{x}{2} + \frac{1}{2}) = abs(x),$$

resulting in

$$-(\frac{x}{2} + \frac{1}{2}) = x \vee \frac{x}{2} + \frac{1}{2} = x.$$

So, if the first character is an a then we need weights $x = \frac{1}{3} \vee x = -1$ and if the first character is a b then we need weights $x = -\frac{1}{3} \vee x = 1$.

Looking at Example 2.60, we observe that in an alphabet of two characters, we need the next signed lexicographical weight to be $\frac{1}{3}$ or -1, if the weight of the current character is negative, for both characters to be equal. We need the next signed lexicographical weight to be $-\frac{1}{3}$ or 1, if the weight of the current character is negative, so that both characters have the same weight.

To obtain these values, we need to have an infinite string, but infinite strings do not exist. Unfortunately, since we are working with computers and these fractions are saved using *floats* this problem also happens when a floating point number (IEEE 754 double) is no longer precise enough. $\frac{1}{3}$ cannot be stored exactly in a float, since we are working with bits. Therefore, in double precision, after 27 characters of $(ab)^n$ rounding makes a weight that is saved as the equivalent value that is saved for $\frac{1}{3}$.

2.5.2 Absolute Lyndon factorization (suffix arrays)

Since the algorithm designed by Wols [Wol26] reveals a problem in a computer with a limited amount of bits to store an integer, we should look for a different algorithm which is also able to create the absolute Lyndon factorization. A possible algorithm is presented by De Mol [dM21] and uses a suffix array. First we look for the two smallest absolute Lyndon factors in the suffix array. After this we compare both absolute Lyndon factors and take the signed lexicographically smallest factor. Then we look for the next two smallest absolute Lyndon factors, with length greater than the removed factors, which is also a suffix of the input string. We compare those two factors and again take the signed lexicographically smallest factor, etc. As De Mol showed, a suffix array is a practical data structure for this factorization. The De Mol algorithm is shown in Algorithm 2.9. This factorization is, as proven by De Mol, computed in linear time [dM21].

This algorithm might be more useful to use than the algorithm introduced by Wols [Wol26] since it does not use very big or very small integers that are not possible to store on a computer. On the other hand, it uses suffix arrays, which are created by complex algorithms if they should be created in linear time.

2.5.3 Bijective Burrows-Wheeler transform using absolute Lyndon words

The BBWT is a transformation over a string x using a totally ordered alphabet Σ and the Lyndon factorization of the string x. Since absolute Lyndon words are closely related to Lyndon words and also use a totally ordered alphabet, it is interesting to observe what happens to the BBWT when we change the Lyndon words to absolute Lyndon words.
There are many possibilities to do this change, such as also changing the complete order on which we sort all the rotations of the absolute Lyndon words, but first, we look into what happens if we consider only the change of Lyndon words to absolute Lyndon words. At first glance, this might give an interesting new transform, but it turns out that this new transform is not bijective.

The absolute bijective Burrows-Wheeler transform (ABBWT) uses steps similar to the BBWT. First, it creates a unique factorization of the input string, after which we generate all the rotations of this string and select the last character of each rotation. This must be done using a totally ordered alphabet. The absolute Lyndon factorization uses the signed-lexicographic order (Definition 2.47), which is totally ordered. Therefore, this seems to satisfy the requirement, but since absolute Lyndon words make use of absolute values of a string, this implies that that in this factorization the strings are not totally ordered.

Example 2.61. Let $\Sigma = \{a, b\}$ with $a < \varepsilon < b$ and $\overline{b} = a$, then using Definition 2.52 the string $y = baab$ and string $x = abba$ have equal signed-lexicographic weight.

This property of absolute Lyndon words is the reason why the ABBWT is not injective. To prove this, we first want to show a new property of absolute Lyndon words.

Theorem 2.62. Let x be a string that is an absolute Lyndon word, then the negative $\overline{x}$ is also an absolute Lyndon word.

Proof. According to Definition 2.52, an absolute Lyndon word is minimal among the absolute values of its suffixes. A string x has the same absolute value as its negative $\overline{x}$ according to Definition 2.51.

Algorithm 2.9: [dM21] Absolute Lyndon factorization (suffix arrays)

Input: A string x of length n

Output: An array of absolute Lyndon words D such that $x = \sum_i D[i]$

```
   // Step 1: initialize
 1 Let S be a suffix array of x
 2 r ← 1
 3 while r ≤ n and x[S[r]] < ε do
 4 |  r ← r + 1                          // r is the index of the weakest positive suffix
 5 ℓ ← r − 1                             // ℓ is the index of the weakest negative suffix
 6 m ← n
 7 i ← 1                                                          // iteration counter
   // Step 2: decomposition
 8 while m > 0 do
   |  // Determine s, the weakest suffix index in x[0...m − 1]
 9 |  if ℓ < 1 then
10 |  |  s ← S[r]
11 |  else
12 |  |  if r > n then
13 |  |  |  s ← S[ℓ]
14 |  |  else
15 |  |  |  if abs(x[S[r]...m]) < abs(x[S[ℓ]...m]) then
16 |  |  |  |  s ← S[r]
17 |  |  |  else
18 |  |  |  |  s ← S[ℓ]
19 |  D[i] ← x[s...m]
20 |  m ← s − 1
   |  // Update ℓ and r
21 |  while ℓ ≥ 1 and S[ℓ] ≥ m do
22 |  |  ℓ ← ℓ − 1
23 |  while r ≤ n and S[r] ≥ m do
24 |  |  r ← r + 1
25 |  i ← i + 1
```

Therefore, all suffixes of x and $\overline{x}$ of the same length have the same absolute value. This means that if x is minimal among the absolute value of its suffixes, then $\overline{x}$ must also be minimal among the absolute value of its suffixes. Thus, if x is an absolute Lyndon word, then $\overline{x}$ is also an absolute Lyndon word. □

This theorem leads to the observation that if a word is an absolute Lyndon word and the negative of this word is also absolute Lyndon and a rotation of the original word, then the ABBWT of both strings is equal.

Theorem 2.63. Let $x \in \{a, b\}^*$, with $a < \varepsilon < b$ and $\overline{b} = a$, be an absolute Lyndon word with $\overline{x} = \rho_k(x)$ for some k. Then $\text{ABBWT}(x) = \text{ABBWT}(\overline{x})$.

Proof. let $x = x_1 x_2 \dots x_n$ and $\overline{x} = \overline{x_1}\ \overline{x_2} \dots \overline{x_n}$. Since $\overline{x} = \rho_k(x)$ we know that

$$\{\rho_k(x) : 0 \le k < n\} = \{\rho_k(\overline{x}) : 0 \le k < n\}.$$

Therefore, the sorted lists of all rotations of both x and $\overline{x}$ are equal, resulting in the same last column. Thus,

$$\text{ABBWT}(x) = \text{ABBWT}(\overline{x}), \quad \text{if } \overline{x} = \rho_k(x). \qquad \square$$

Example 2.64. Let $x = abba$, an absolute Lyndon word, then $\overline{x} = baab$. Then $baab = \rho_2(abba)$. Both have the same sorted list of rotations

$$\text{sort}(\bigcup_{k=1}^{n}\{\rho_k(abba)\}) = \text{sort}(\bigcup_{k=1}^{n}\{\rho_k(baab)\}) = \{aabb, abba, baab, bbaa\}.$$

Thus,

$$\text{ABBWT}(x) = \text{ABBWT}(\overline{x}) = baba.$$

Theorem 2.63 proves that the ABBWT is not injective, since some strings in the domain map to the same string in the codomain. Although the ABBWT not being injective is a reason to not look further into the transform, we know that strings very rarely exist out of just one absolute Lyndon word. Since strings often consists of multiple absolute Lyndon words, we want to look at the output of the transform of two strings with equal absolute Lyndon words, except for one that is the negative of the original absolute Lyndon word.

Theorem 2.65. let $x = x_1 x_2 \dots x_k \dots x_n$ and $y = y_1 y_2 \dots y_k \dots y_n$, with x_i and y_i absolute Lyndon words. Suppose $y_i = x_i$, where $i \neq k$, and $y_k = \rho_d(\overline{x_k})$ for some d. Then $\text{ABBWT}(x) = \text{ABBWT}(y)$.

Proof. We know that

$$\bigcup_{i=1}^{n}\{\rho_\ell(x_i) : 0 \le \ell < \text{len}(x_i)\} = \bigcup_{i=1}^{n}\{\rho_\ell(y_i) : 0 \le \ell < \text{len}(y_i)\},$$

Since

$$\{\rho_\ell(x_i) : 0 \le \ell < \text{len}(x_i)\} = \{\rho_\ell(y_i) : 0 \le \ell < \text{len}(y_i)\} \text{ for all } i = 1, \dots, n.$$

This is true since for $i \neq k$ we have the exact same rotations because both x_i and y_i are equal for this condition. If $i = k$ then we know that both have the exact same rotations because of Theorem 2.63. $\square$

Example 2.66. Let $x = bbaababb$ and $y = babbaabb$. The absolute Lyndon factorization returns the absolute Lyndon factors $x = \{b, baab, abb\}$ and $y = \{b, abba, abb\}$. It is obvious that $x_1 = y_1$, $x_3 = y_3$, and $y_2 = \rho_0(\overline{x_2})$. This results in

$$\bigcup_{i=1}^{3}\{\rho_\ell(x_i) : 0 \le \ell < \text{len}(x_i)\} = \{b, aabb, abba, bbaa, baab, abb, bab, bba\},$$

and

$$\bigcup_{i=1}^{3}\{\rho_\ell(y_i) : 0 \le \ell < \mathrm{len}(y_i)\} = \{b, bbaa, baab, aabb, abba, abb, bab, bba\},$$

where the colors indicate which rotation is a rotation of which Lyndon word. We observe that both have exactly the same rotations; therefore, both input strings must have the same ABBWT.

3 Experiments

In this section, we evaluate the effect of character orderings for the BBWT on compression performance. The goal is to measure how the transformation choice influences the compression ratio within the same pipeline. We compare the BBWT with randomly selected orderings on three files from the Pizza & Chili corpus [FN05] and one file from the Calgary corpus [Pow01], using Nelson's reference implementation [Nel96].

3.1 Setup

For the experiments, the algorithm was implemented in `C++`. We are interested in the effect that alphabet reordering has on the compression ratio. Some options were considered to compare the effect of the transformation:

- The first option is to count all the runs of the same character in the transform, and the transformation with fewer runs should have a better compression performance. For example, we have string $x = aabbaa$ and string $y = aaabbb$ then string x has three runs of characters (aa, bb, aa) and string y only has two (aaa, bbb). Therefore, string y has a better performance. The problem with this setup for the comparison is that the BWT and BBWT are often used together with other algorithms. This implementation would return some insight into the effect of the transformation, but is not able to return a reliable result for a real-world use case.

- The second option is to make use of Nelson's reference implementation [Nel96]. This implementation carries out the compression in five steps:

  (i) initial run-length encoding

  (ii) Burrows-Wheeler transform

  (iii) move-to-front

  (iv) another run-length encoding

  (v) arithmetic coding

  This implementation is quite similar to the way the *bzip*2 algorithm works [Sew01].

Since we are interested in the compression performance when using different orderings, we decided to use the second option. Although leaving out the initial run-length encoding results in a better compression performance, we leave this in for better comparison with other papers.
Nelson's reference implementation calculates the BBWT over a block of 200.000 bytes. This is an understandable choice since RAM is not infinite. Therefore, compressing large files would be almost impossible, since the entire file should be accessible in RAM. For our results we are interested in the number of Lyndon words of a string compared to the compression ratio. Therefore we will calculate the compression pipeline over the entire file, otherwise calculations over large files would reveal similar results to smaller files.

It is also important to note that the cost of calculation for this algorithm is very high. This leads to the issue that we cannot generate enough results to have a statistically significant conclusion. Therefore we will only look at four selected files with different properties.

3.1.1 Dataset

For testing we will use a few different corpora obtained from different places. First of all we will use the Calgary corpus [BWC89]. This corpus was developed in the late 1980s, and during the 1990s became the standard for lossless compression evaluation. The collection is a little outdated, but still reliable for compression performance indication. The Calgary corpus has 14 files (originally 18, but paper3, paper4, paper5, and paper6 are removed from the corpus because they do not add to the evaluation, since they are highly similar to the other paper files.) and consists of different data types such as source code in some languages, papers, geophysical data, and two books as shown in Table 3.1. This corpus is also used as a benchmark in the paper [GS09]. From this corpus we will only use the first file "bib" since it is a small file with mostly characters from the Latin alphabet. Another corpus that we considered was the Canterbury corpus [AB97]. This corpus

Table 3.1: Calgary corpus files.

File	Abbrev	Category	Size (in bytes)
bib	bib	Bibliography (refer format)	111.261
book1	book1	Fiction book	768.771
book2	book2	Non-fiction book (troff format)	610.856
geo	geo	Geophysical data	102.400
news	news	USENET batch file	377.109
obj1	obj1	Object code for VAX	21.504
obj2	obj2	Object code for Apple Mac	246.814
paper1	paper1	Technical paper	53.161
paper2	paper2	Technical paper	82.199
pic	pic	Black and white fax picture	513.216
progc	progc	Source code in "C"	39.611
progl	progl	Source code in LISP	71.646
progp	progp	Source code in PASCAL	49.379
trans	trans	Transcript of terminal session	93.695

was created with the concern that algorithms were being fine-tuned to the Calgary corpus. The Canterbury corpus consists of 11 files, with written English texts, HTML source code, spreadsheets, and other data types and is shown in Table 3.2. Since the cost of calculation of the BBWT is too high for many files, we decided not to use this corpus, since it is closely related to the Calgary corpus.

We also found other corpora such as "The Artificial Corpus" and "The large Corpus" [Pow01]. These two corpora will not be very useful since the Artificial Corpus only consists of the same characters repeated over and over and will therefore not show a very useful result. We will not use the Large Corpus since it is mostly used as a benchmark for methods that require larger files to measure speed, but since we will not be focusing on speed, this is not very interesting to us.
A newer much larger dataset that we will also be using is the "Pizza & Chili Corpus" [FN05] as shown in Table 3.3. This dataset is much larger and therefore interesting for measuring the effect of the BBWT on larger files. It also makes a distinction between files with different alphabet sizes, which is also interesting for comparing different outcomes.

Table 3.2: Canterbury corpus files.

File	Abbrev	Category	Size (in bytes)
alice29.txt	text	English text	152.089
asyoulik.txt	play	Shakespeare	125.179
cp.html	html	HTML source	24.603
fields.c	Csrc	C source	11.150
grammar.lsp	list	LISP source	3.721
kennedy.xls	Excl	Excel Spreadsheet	1.029.744
lcet10.txt	tech	Technical writing	426.754
plrabn12.txt	poem	Poetry	481.861
ptt5	fax	CCITT test set	513.216
sum	SPRC	SPARC Executable	38.240
xargs.1	man	GNU manual page	4.227

Table 3.3: Pizza & Chili Corpus files.

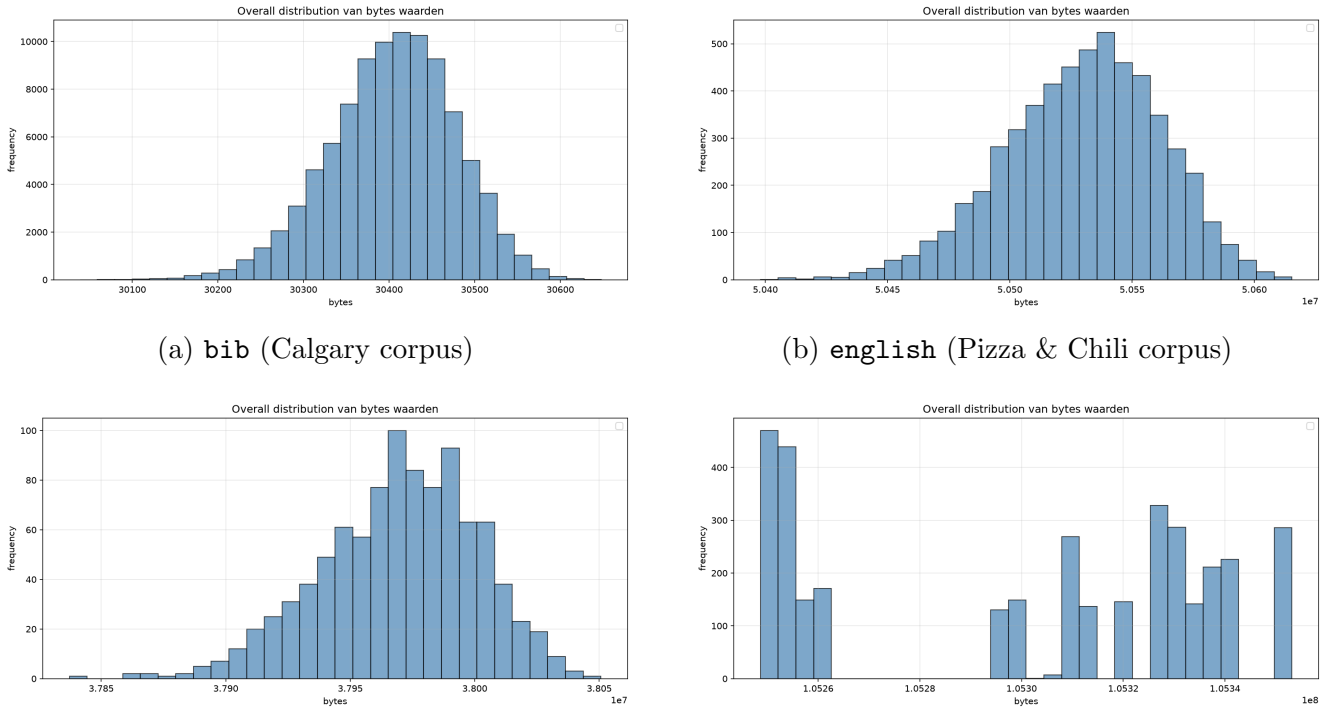
Collection	Size (bytes)	Alphabet size
SOURCES	210,866,607	230
PITCHES	55,832,855	133
PROTEINS	1,184,051,855	27
DNA	403,927,746	16
ENGLISH	2,210,395,553	239
XML	294,724,056	97

4 Results

This section presents the results of our experiments on the compression pipeline introduced by Nelson [Nel96] under varying alphabet orderings. We focus on two perspectives. First, we characterize the distribution of compressed sizes obtained from randomly generated orderings for each dataset, which provides insight into typical performance and variability. Second, we relate the achieved compression ratio to the number of Lyndon factors induced by each ordering, in order to assess whether factorization complexity correlates with compression effectiveness.

4.1 Distribution of bytes

In this subsection, we analyze the overall distribution of the number of bytes after compression and how often they occur in a randomly generated set of orders. For *bib* (1a), *english* (1b), and *sources* (1c), the distribution exhibits a pronounced mode, indicating that many random orderings yield similar compressed sizes. For *sources*, however, the behavior near the peak is less clear. the irregularities may reflect sampling noise, but they could also indicate structure in the data that is not yet captured due to the current number of random orderings. *dna* (1d) leads to a different distribution than the other files. It seems that this file has more clustered groupings. Although more clusters could exist, for now we assume only two clusters, below $1.0528 * 10^8$ and above $1.0528 * 10^8$.



(a) `bib` (Calgary corpus)

(b) `english` (Pizza & Chili corpus)

(c) `sources` (Pizza & Chili corpus)

(d) `dna` (Pizza & Chili corpus)

Figure 1: Distribution of bytes and their frequency for four files from the Calgary and Pizza & Chili corpora.

To analyze this split in *dna*, we did some further experiments to try and find out if we can find a reason for this split. First we found out that under our random sampling, an order has a chance of 34.6% of being in the lower group. The file consists of sixteen different characters, but since five of those ($A, C, G, T, \backslash n$ (newline)) appeared more often than all the others, we wanted to see if these five could explain the difference in order. First, we looked for a pairwise explanation. The results are shown in Table 4.1.
We cannot see a clear pairwise distribution from this table, thus, we looked if the order of these five characters could always place an order in the lower or higher group. From Table A.1 we can interpret that these five characters can place an order in the higher or lower half of the distribution.

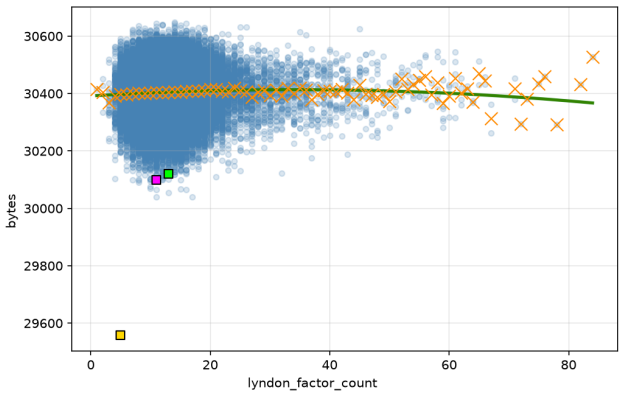
4.2 Distribution of bytes and number of Lyndon factors

Next, we plotted the amount of bytes after compression against the Lyndon factor count of the file for each randomly selected order. For the plot of *sources* (2c) we decided to remove three data points that had very high Lyndon factor counts. We did this because these points made the plot unreadable and had a big influence on the trend line. Unfortunately as all the plots in Figure 2 show, all trend lines are almost flat. Therefore it is hard to say if there exists an optimal amount of Lyndon factors for certain data structures. For each file we also plotted three specific orderings: ASCII, MFS and LFS. We can see that MFS and LFS are on the lower side in each scatterplot (except for *dna*). What is much more noticable is that the ASCII ordering is much lower then every point in the cluster (again except for *dna*). As mentioned before, the file *dna* is different from the

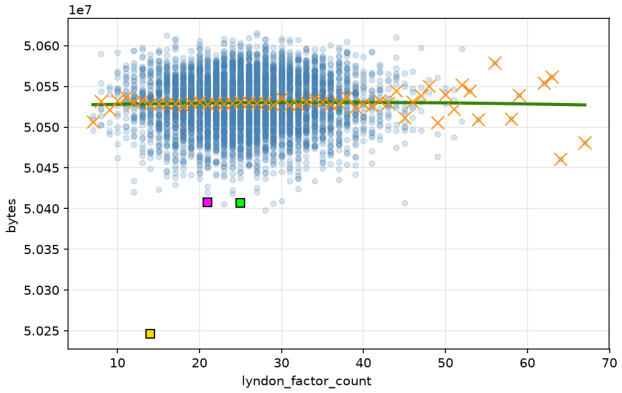
Table 4.1: Pairwise relation frequencies for the low and high groups, with the difference in percentage points (high minus low) and the group in which the relation is stronger.

Relation	Low (%)	(n)	High (%)	(n)	Δ (percentage point)	Stronger in
A<G	53.13	(1229)	49.07	(2319)	−4.06	low
A< \n	52.48	(1229)	49.03	(2319)	−3.45	low
A<T	50.94	(1229)	49.12	(2319)	−1.82	low
A<C	50.94	(1229)	49.46	(2319)	−1.47	low
C<G	49.06	(1229)	50.19	(2319)	1.13	high
C< \n	50.77	(1229)	49.85	(2319)	−0.92	low
C<T	49.96	(1229)	50.58	(2319)	0.62	high
G< \n	49.39	(1229)	49.94	(2319)	0.55	high
T< \n	49.15	(1229)	49.68	(2319)	0.53	high
G<T	50.94	(1229)	50.84	(2319)	−0.09	low

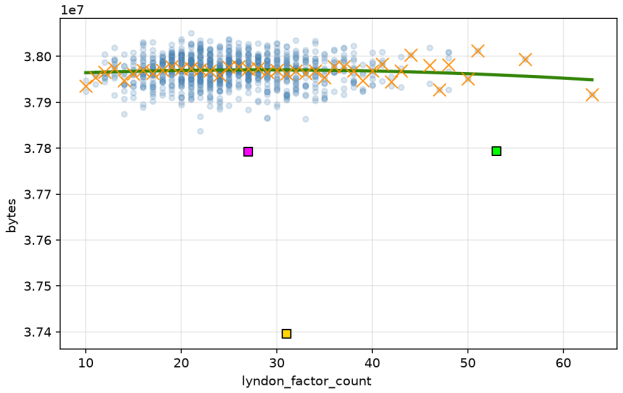
others. First of all, the ASCII, MFS, and LFS orders are not even in the lower group of the plot; besides that there is reason to assume that there exist at least two groups in this order. Therefore, we have plotted three trend lines, one common trend line for all data points, and two lines for the larger and smaller groups.



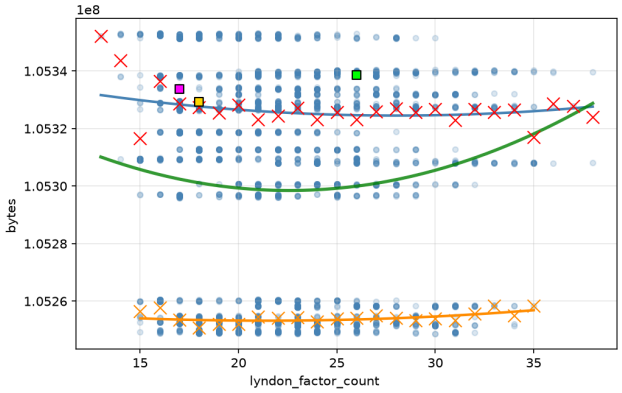
(a) `bib`



(b) `english`



(c) `sources`



(d) `dna`

Figure 2: Scatterplots of different orderings of the characters for four datasets. On the horizontal axis we see the number of Lyndon factors in a string for a certain ordering, on the vertical axis we have the number of bytes after compression. The orange X marks the average number of Lyndon factors, the green line is the trend line over all points. We also see a green square for the LFS order, the magenta is the MFS order, and the yellow is the ASCII order.

5 Conclusions and Further Research

The original question that we tried to answer in this paper is:

> *What is the effect of changing Lyndon words to absolute Lyndon words in the bijective Burrows-Wheeler transform with respect to compression performance?*

This research question turned out to be not fully answerable, but we were able to answer a part of the research question. The effect of changing Lyndon words to absolute Lyndon words in the BBWT is that the BBWT is no longer bijective.

The reconstructed research question that we tried to answer is the following:

What is the effect of alphabet reordering on the bijective Burrows-Wheeler transform with respect to compression performance?

First we discuss our subquestions. Our first subquestion is: "*What is the effect of the number of Lyndon factors of a string on the BBWT?*". As Figure 2 shows, we notice that there is little effect of the number of Lyndon factors. The trend line is close to horizontal, and the difference between the maximum and minimum is not large enough to be statistically significant without more results.

Our second subquestion is: "*What alphabet order maximizes the effect of the BBWT?*". Figure 2 indicates that for most datasets, standard ASCII ordering yields better compression results than MFS, LFS, and randomly sampled orderings. Although MFS and LFS orderings also perform competitively, they do not exceed ASCII.
The *dna* dataset constitutes an exception. For this file, several orderings achieve better compression than ASCII, suggesting that the ordering that benefits BBWT most is data dependent. Although we cannot provide a definitive explanation, a hypothesis is that this behavior is driven by the underlying structure and reduced alphabet of DNA sequences. In contrast to the other datasets, which mostly consist of Latin alphabet text and a broader range of punctuation and control characters (*bib* uses 81 distinct bytes; *english* 239; and *sources* 230), the *dna* file uses only 16 characters. Because ASCII's code-point ordering is historically optimized for text-like data with large, structured character classes, it may align well with the character distributions and local context in *bib*, *english*, and *sources*, but not with the smaller and differently structured DNA alphabet. This mismatch may explain why ASCII is not among the best-performing orderings for *dna*.
As shown in Section 4.2 there is strong evidence that for DNA sequences there do exist orderings that result in better compression ratios. For future work, it would be interesting to see if these orders reveal better compression for all DNA sequences or if this differs per sequence. If strings based on the Latin alphabet have a best way to be ordered, as our results suggest is the ASCII order, then such an order and encoding might also exist for DNA sequences. Since DNA characters have an opposite character, a structure similar to the absolute Lyndon words might reveal a version of the BBWT where characters can still have a negative value while still meeting the requirement of being totally ordered.
For this thesis we only looked at the effect of alphabet reordering for the BBWT. Since this thesis got results using Nelson's reference implementation, for future research it might also be interesting to look into the effect of alphabet reordering for the other algorithms used in this reference implementation. This would arguably change the results, especially when using fewer characters in an alphabet, from which algorithms like MTF and ARI would benefit greatly.
In conclusion, this research provides evidence that alphabet reordering has an effect on the compression performance of the BBWT. Although the results reveal strong evidence for this effect, we do not have enough data to prove that this effect is statistically significant. We observe that ASCII might already be the best order for most data types, but more research is needed to find the perfect order for data types such as DNA.

References

[AB97] R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In *Proceedings of the Data Compression Conference (DCC)*, pages 201–210, Snowbird, Utah, USA, 1997. IEEE Computer Society.

[AL23] M. K. Albertini and F. A. Louza. Practical evaluation of Lyndon factors via alphabet reordering. *Mathematics*, 11(1):139, 2023.

[BKKP21] H. Bannai, J. Kärkkäinen, D. Köppl, and M. Piatkowski. Constructing the bijective and the extended Burrows–Wheeler Transform in linear time. In *Proceedings of the 32nd Annual Symposium on Combinatorial Pattern Matching*, 2021.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report SRC Research Report 124, Digital Equipment Corporation, Systems Research Center, 1994.

[BWC89] T. C. Bell, I. H. Witten, and J. G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–591, December 1989.

[CFL58] K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus, iv. the quotient groups of the lower central series. *Annals of Mathematics*, 68(1):81–95, 1958.

[dM21] T. de Mol. Synchronized cherries, July 2021. Bachelor thesis, Leiden University, Thesis supervisors: M.J.H. van den Bergh, W.A. Kosters.

[Duv83] J. P. Duval. Factorizing words over an ordered alphabet. *Journal of Algorithms*, 4(4):363–381, 1983.

[FN05] P. Ferragina and G. Navarro. Pizza&chili corpus: Compressed indexes and their testbeds. https://pizzachili.dcc.uchile.cl/index.html, September 2005.

[GRS07] R. Giancarlo, A. Restivo, and M. Sciortino. From first principles to the Burrows and Wheeler Transform and beyond, via combinatorial optimization. *Theoretical Computer Science*, 387(3):236–248, 2007.

[GS09] J. Gil and D. Allen Scott. A bijective string sorting transform, July 2009. Google, Inc., Haifa, Israel; El Paso, Texas, USA.

[GS12] J. Y. Gil and D. A. Scott. A bijective string sorting transform. https://doi.org/10.48550/arxiv.1201.3077, 2012. arXiv:1201.3077.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952.

[Lyn54] R. C. Lyndon. On burnside's problem. *Transactions of the American Mathematical Society*, 77(2):202–215, 1954.

[Nel96] M. Nelson. Data compression with the Burrows-Wheeler transform. *Dr. Dobb's Journal*, September 1996. Available online: https://drdobbs.com/data-compression-with-the-burrows-wheeler-transform/.

[NZC09] G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting, 2009.

[Pow01] M. Powell. Descriptions of the corpora. The Canterbury Corpus, University of Canterbury, January 2001.

[Sew01] J. Seward. *bzip2 and libbzip2, version 1.0.8: A Program and Library for Data Compression*. sourceware.org, July 2001.

[WNC87] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.

[Wol26] R. Wols. Synchronous combinatorial games. Master thesis, Universiteit Utrecht, 2026.

A Results

Table A.1: Signature frequencies in the low and high groups, including totals and the difference in percentage points (high minus low).

Signature	Total	Low % (count)	High % (count)	Δ (pp)
A<G<NL<T<C	45	3.66% (45)	0.00% (0)	-3.66
C<T<NL<A<G	43	3.50% (43)	0.00% (0)	-3.50
G<A<T<C<NL	39	3.17% (39)	0.00% (0)	-3.17
NL<G<A<T<C	38	3.09% (38)	0.00% (0)	-3.09
A<NL<G<C<T	38	3.09% (38)	0.00% (0)	-3.09
C<T<A<G<NL	38	3.09% (38)	0.00% (0)	-3.09
C<T<A<NL<G	38	3.09% (38)	0.00% (0)	-3.09
T<C<NL<A<G	36	2.93% (36)	0.00% (0)	-2.93
C<NL<T<G<A	35	2.85% (35)	0.00% (0)	-2.85
A<NL<G<T<C	34	2.77% (34)	0.00% (0)	-2.77
NL<C<T<G<A	34	2.77% (34)	0.00% (0)	-2.77
A<G<C<NL<T	34	2.77% (34)	0.00% (0)	-2.77
A<G<T<NL<C	32	2.60% (32)	0.00% (0)	-2.60
C<NL<T<A<G	32	2.60% (32)	0.00% (0)	-2.60
T<C<A<G<NL	32	2.60% (32)	0.00% (0)	-2.60
NL<A<G<T<C	32	2.60% (32)	0.00% (0)	-2.60
G<A<NL<C<T	32	2.60% (32)	0.00% (0)	-2.60
A<G<C<T<NL	31	2.52% (31)	0.00% (0)	-2.52
NL<T<C<G<A	31	2.52% (31)	0.00% (0)	-2.52
T<C<A<NL<G	30	2.44% (30)	0.00% (0)	-2.44
G<A<T<NL<C	30	2.44% (30)	0.00% (0)	-2.44
NL<T<C<A<G	30	2.44% (30)	0.00% (0)	-2.44
T<C<NL<G<A	29	2.36% (29)	0.00% (0)	-2.36
G<A<C<NL<T	29	2.36% (29)	0.00% (0)	-2.36
A<G<NL<C<T	29	2.36% (29)	0.00% (0)	-2.36
C<T<G<NL<A	28	2.28% (28)	0.00% (0)	-2.28
A<G<T<C<NL	28	2.28% (28)	0.00% (0)	-2.28
G<A<C<T<NL	27	2.20% (27)	0.00% (0)	-2.20
C<T<G<A<NL	27	2.20% (27)	0.00% (0)	-2.20
T<C<G<A<NL	27	2.20% (27)	0.00% (0)	-2.20
G<NL<A<T<C	27	2.20% (27)	0.00% (0)	-2.20
T<NL<C<G<A	27	2.20% (27)	0.00% (0)	-2.20
NL<A<G<C<T	26	2.12% (26)	0.00% (0)	-2.12
G<NL<A<C<T	25	2.03% (25)	0.00% (0)	-2.03
G<A<NL<T<C	25	2.03% (25)	0.00% (0)	-2.03
NL<G<A<C<T	25	2.03% (25)	0.00% (0)	-2.03
NL<C<T<A<G	24	1.95% (24)	0.00% (0)	-1.95

Continued on next page

Signature	Total	Low % (count)	High % (count)	Δ (pp)
T<C<G<NL<A	22	1.79% (22)	0.00% (0)	-1.79
T<NL<C<A<G	21	1.71% (21)	0.00% (0)	-1.71
C<NL<G<T<A	39	0.00% (0)	1.68% (39)	1.68
T<G<A<NL<C	39	0.00% (0)	1.68% (39)	1.68
T<G<NL<A<C	38	0.00% (0)	1.64% (38)	1.64
T<G<A<C<NL	38	0.00% (0)	1.64% (38)	1.64
NL<C<A<G<T	37	0.00% (0)	1.60% (37)	1.60
T<A<NL<G<C	37	0.00% (0)	1.60% (37)	1.60
C<A<NL<G<T	36	0.00% (0)	1.55% (36)	1.55
G<C<T<NL<A	36	0.00% (0)	1.55% (36)	1.55
C<T<NL<G<A	19	1.55% (19)	0.00% (0)	-1.55
G<T<NL<A<C	35	0.00% (0)	1.51% (35)	1.51
NL<C<G<T<A	35	0.00% (0)	1.51% (35)	1.51
A<C<G<T<NL	35	0.00% (0)	1.51% (35)	1.51
T<NL<A<C<G	35	0.00% (0)	1.51% (35)	1.51
NL<T<G<C<A	34	0.00% (0)	1.47% (34)	1.47
NL<G<T<C<A	34	0.00% (0)	1.47% (34)	1.47
A<T<G<C<NL	34	0.00% (0)	1.47% (34)	1.47
G<C<A<NL<T	34	0.00% (0)	1.47% (34)	1.47
G<NL<T<C<A	33	0.00% (0)	1.42% (33)	1.42
NL<A<C<G<T	33	0.00% (0)	1.42% (33)	1.42
C<NL<A<T<G	33	0.00% (0)	1.42% (33)	1.42
G<C<NL<T<A	33	0.00% (0)	1.42% (33)	1.42
NL<G<T<A<C	32	0.00% (0)	1.38% (32)	1.38
T<A<NL<C<G	32	0.00% (0)	1.38% (32)	1.38
C<G<A<T<NL	31	0.00% (0)	1.34% (31)	1.34
A<T<NL<G<C	31	0.00% (0)	1.34% (31)	1.34
T<G<NL<C<A	31	0.00% (0)	1.34% (31)	1.34
G<NL<T<A<C	31	0.00% (0)	1.34% (31)	1.34
C<G<A<NL<T	31	0.00% (0)	1.34% (31)	1.34
C<A<T<NL<G	31	0.00% (0)	1.34% (31)	1.34
C<NL<A<G<T	31	0.00% (0)	1.34% (31)	1.34
NL<C<A<T<G	31	0.00% (0)	1.34% (31)	1.34
A<NL<T<C<G	31	0.00% (0)	1.34% (31)	1.34
C<G<T<A<NL	31	0.00% (0)	1.34% (31)	1.34
A<T<G<NL<C	30	0.00% (0)	1.29% (30)	1.29
NL<A<T<C<G	30	0.00% (0)	1.29% (30)	1.29
NL<G<C<T<A	30	0.00% (0)	1.29% (30)	1.29
G<T<C<A<NL	30	0.00% (0)	1.29% (30)	1.29
G<C<NL<A<T	30	0.00% (0)	1.29% (30)	1.29
G<T<C<NL<A	30	0.00% (0)	1.29% (30)	1.29

Continued on next page

Signature	Total	Low % (count)	High % (count)	Δ (pp)
NL<T<A<C<G	30	0.00% (0)	1.29% (30)	1.29
A<C<NL<G<T	30	0.00% (0)	1.29% (30)	1.29
C<G<NL<A<T	30	0.00% (0)	1.29% (30)	1.29
T<NL<G<A<C	30	0.00% (0)	1.29% (30)	1.29
A<C<T<G<NL	30	0.00% (0)	1.29% (30)	1.29
NL<C<G<A<T	29	0.00% (0)	1.25% (29)	1.25
A<NL<C<G<T	29	0.00% (0)	1.25% (29)	1.25
C<A<T<G<NL	28	0.00% (0)	1.21% (28)	1.21
A<C<NL<T<G	28	0.00% (0)	1.21% (28)	1.21
A<C<G<NL<T	28	0.00% (0)	1.21% (28)	1.21
NL<A<C<T<G	27	0.00% (0)	1.16% (27)	1.16
G<T<A<C<NL	27	0.00% (0)	1.16% (27)	1.16
T<G<C<A<NL	27	0.00% (0)	1.16% (27)	1.16
T<NL<G<C<A	27	0.00% (0)	1.16% (27)	1.16
T<A<G<NL<C	27	0.00% (0)	1.16% (27)	1.16
G<NL<C<T<A	26	0.00% (0)	1.12% (26)	1.12
T<G<C<NL<A	26	0.00% (0)	1.12% (26)	1.12
A<NL<C<T<G	26	0.00% (0)	1.12% (26)	1.12
A<C<T<NL<G	26	0.00% (0)	1.12% (26)	1.12
C<G<T<NL<A	26	0.00% (0)	1.12% (26)	1.12
G<C<A<T<NL	26	0.00% (0)	1.12% (26)	1.12
T<A<C<NL<G	25	0.00% (0)	1.08% (25)	1.08
C<A<G<T<NL	25	0.00% (0)	1.08% (25)	1.08
T<A<C<G<NL	25	0.00% (0)	1.08% (25)	1.08
NL<A<T<G<C	25	0.00% (0)	1.08% (25)	1.08
G<C<T<A<NL	25	0.00% (0)	1.08% (25)	1.08
NL<T<A<G<C	24	0.00% (0)	1.03% (24)	1.03
A<T<C<NL<G	24	0.00% (0)	1.03% (24)	1.03
G<NL<C<A<T	24	0.00% (0)	1.03% (24)	1.03
T<A<G<C<NL	24	0.00% (0)	1.03% (24)	1.03
C<A<G<NL<T	24	0.00% (0)	1.03% (24)	1.03
C<G<NL<T<A	23	0.00% (0)	0.99% (23)	0.99
C<NL<G<A<T	23	0.00% (0)	0.99% (23)	0.99
A<T<NL<C<G	23	0.00% (0)	0.99% (23)	0.99
NL<G<C<A<T	22	0.00% (0)	0.95% (22)	0.95
A<T<C<G<NL	22	0.00% (0)	0.95% (22)	0.95
C<A<NL<T<G	21	0.00% (0)	0.91% (21)	0.91
NL<T<G<A<C	20	0.00% (0)	0.86% (20)	0.86
T<NL<A<G<C	20	0.00% (0)	0.86% (20)	0.86
A<NL<T<G<C	20	0.00% (0)	0.86% (20)	0.86
G<T<NL<C<A	19	0.00% (0)	0.82% (19)	0.82

Continued on next page

Signature	Total	Low % (count)	High % (count)	Δ (pp)
G<T<A<NL<C	16	0.00% (0)	0.69% (16)	0.69