Abstract. The process of sorting the suffixes of a text plays a fundamental role in Text Algorithms. They are used for instance in the constructions of the Burrows-Wheeler transform and the suffix array, widely used in several fields of Computer Science. For this reason, several recent researches have been devoted to finding new strategies to obtain effective methods for such a sorting. In this paper we introduce a new methodology in which an important role is played by the Lyndon factorization, so that the local suffixes inside factors detected by this factorization keep their mutual order when extended to the suffixes of the whole word. This property suggests a versatile technique that easily can be adapted to different implementative scenarios.

Keywords: sorting suffixes, BWT, suffix array, Lyndon words, Lyndon factorization

1 Introduction

The sorting of the suffixes of a text plays a fundamental role in Text Algorithms with several applications in many areas of Computer Science and Bioinformatics. For instance, it is a fundamental step, in implicit or explicit way, for the construction of the suffix array (SA) and the Burrows-Wheeler Transform (bwt). The SA, introduced in 1990 (cf. [19]), is a sorted array of all suffixes of a string, where the suffixes are identify by using their positions in the string. Several strategies that privilege the efficiency of the running time or the low memory consumption have been widely investigated (cf. [22,16]). The bwt, introduced in 1994 (cf. [6]), permutes the letters of a text according to the sorting of its cyclic rotations, making the text more compressible (cf. [4]). A recent survey on the combinatorial properties that guarantee such a compressibility after the application of bwt can be found in [24] (cf. also [23]). Moreover, in the last years the SA and the bwt, besides being important tools in Data Compression, have found many applications well beyond its original purpose (cf. [1,13,11,24,12]).

The goal of this paper is to introduce a new strategy for the sorting of the suffixes of a word that opens new scenarios of the computation of the SA and the bwt.

Our strategy uses a well known factorization of a word W called the Lyndon factorization and is based on a combinatorial property proved in this paper, that allows to sort the suffixes of W ("global suffixes") by using the sorting of the suffixes inside each block of the decomposition ("local suffixes").

The Lyndon factorization is based on the fact that any word W can be written uniquely as $W = L_1L_2\cdots L_k$, where

- the sequence $L_1, L_2, \ldots, L_k$ is non-increasing with respect to lexicographic order;
- each $L_i$ is strictly less than any of its proper cyclic shift (Lyndon words).
This factorization was introduced in [7] and a linear time algorithm is due to Duval [11]. The intuition that the knowledge of Lyndon factorization of a text can be used for the computation of the suffix array of the text itself has been introduced in [5]. Conversely, a way to find the Lyndon factorization from the suffix array can be found in [17].

If \( U \) is a factor of a word \( W \) we say that the sorting of the local suffixes of \( U \) is compatible with the sorting of the global suffixes of \( W \) if the mutual order of two local suffixes in \( U \) is kept when they are extended as global suffixes. The main theorem in this paper states that if \( U \) is a concatenation of consecutive Lyndon factors, then the local suffixes in \( U \) are compatible with the global suffixes. This suggests some new algorithmic scenarios for the constructions of the SA and the \textit{bwt}. In fact, by performing the Lyndon factorization of a word \( W \) by Duval’s algorithm, one does not need to get to the end of the whole word in order to start the decomposition into Lyndon factors. Since our result allow to start the sorting of the local suffixes (compatible with the sorting of the global suffixes) as soon as the first Lyndon word is discovered, this may suggest an online algorithm, that do not require to read the entire word to start sorting. Moreover, the independence of the sorting of the local suffixes inside the different Lyndon factors of a text suggests also a possible parallel strategy to sort the global suffixes of the text itself.

In Section 2 we give the fundamental notions and results concerning combinatorics on words, the Lyndon factorization, the Burrows-Wheeler transform and the suffix array. In Section 3 we first introduce the notion of global suffix on a text and local suffix inside a factor of the text. Then we prove the compatibility between the ordering of local suffixes and the ordering of global suffixes. In Section 4 we describe an algorithm that uses the above result to incrementally construct the \textit{bwt} of a text. Such a method can be also used to explicitly construct the \textit{SA} of the text. In Section 5 we discuss about some possible improvements and developments of our method, including implementations in external memory or in place constructions. Finally, we compare our strategy for sorting suffixes with the method proposed in [12] in which a lightweight computation of the \textit{bwt} of a text is performed by partitioning it into factors having the same length.

## 2 Preliminaries

Let \( \Sigma = \{c_1, c_2, \ldots, c_\sigma\} \) be a finite alphabet with \( c_1 < c_2 < \cdots < c_\sigma \). Given a finite word \( W = a_1a_2\cdots a_n, a_i \in \Sigma \) for \( i = 1, \ldots, n \), a factor of \( W \) is written as \( W[i, j] = a_ia_{i+1}\cdots a_j \). A factor \( W[1, j] \) is called a prefix, while a factor \( W[i, n] \) is called a suffix. In this paper, we also denote by \( \text{suf}_W(i) \) as the suffix of \( W \) starting from position \( i \). We omit \( W \) when there is no danger of ambiguity. We say that \( x, y \in \Sigma^* \) are conjugate (or cyclic shift) or \( y \) is a conjugate of \( x \) if \( x = uv \) and \( y = vu \) for some \( u, v \in \Sigma^* \). Recall that conjugacy is an equivalent relation.

A Lyndon word is a primitive word which is also the minimum in its conjugacy class, with respect to the lexicographic order relation. In [18,11], one can find a linear algorithm that for any word \( W \in \Sigma^* \) computes the Lyndon word of its conjugacy class. We call it the Lyndon word of \( W \). Lyndon words are involved in a nice and important factorization property of words.

**Theorem 1.** [7] Every word \( W \in \Sigma^+ \) has a unique factorization \( W = L_1L_2\cdots L_k \) such that \( L_1 \geq_{\text{lex}} \cdots \geq_{\text{lex}} L_k \) is a non-increasing sequence of Lyndon words.
We call this factorization the Lyndon factorization of a word and it can be computed in linear time (see for instance [11,18]). Duval in [11] presents two variants of an algorithm of factorization of a word into Lyndon words in time linear in the length of the word. The first variant of the algorithm uses only three variables for a complete computation and it requires no more than \( 2n \) comparisons between two letters. The second one is slightly faster in that sense that it requires no more than \( \frac{3n}{2} \) comparisons but it uses an auxiliary storage of size \( \frac{n^2}{2} \). The basis idea for both these variants is finding each factor of the decomposition of the word \( W \) from left to right by eventually reading a long enough prefix of the next Lyndon factor.

Lyndon factorization has been realized also in parallel (cf. [3]) and in external memory (cf. [24]).

One way to define the Burrows-Wheeler Transform (\( bwt \)) of a string \( W \) of length \( n \) (although not the most efficient way to compute it) is to construct all \( n \) cyclic shifts of \( W \) and sort them lexicographically. The output of \( bwt \) consists of the pair \((L, I)\), where \( L \) is the sequence of the last character of each rotation in the sorted list and \( I \) is an integer denoting the position of the original word in the list.

Another more efficient way consists in the concatenating at the the input string \( W \) a symbol $ that is smaller than any other letter. In this case, the \( bwt \) is intuitively described as follows: given a word \( W \in \Sigma^* \), \( bwt(W) \) is a word obtained by sorting the list of the suffixes of \( W\$ \) and by concatenating the symbols preceding in \( W \) each suffix in the sorted list. In both the cases, it is an invertible transform, i.e., one can recover the original text from its \( bwt \).

Note that, in general, the sorting of the conjugates of a word \( W \) and the sorting of the suffixes of a word \( W\$ \) is different, but, as consequence of the properties of Lyndon words, when the word \( W \) is the Lyndon word, then the two sorting coincide (cf. [15, Lemma 12]). A study of the combinatorial aspects that connect these two sorting can be found in [5]. In this study an important role is played by the notion of Lyndon word.

Given a text \( W \) of length \( n \), the suffix array (\( SA \)) for \( W \) is an array of integers of range 1 to \( n+1 \) specifying the lexicographic ordering of the suffixes of the string \( W \). It will be convenient to assume that \( W[n+1] = \$ \), where \( \$ \) is smaller than any other letter. That is, \( SA[j] = i \) if and only if \( W[i,n+1] \) is the \( j \)-th suffix of \( W \) in ascending lexicographical order.

![Figure 1](image)

**Figure 1.** The table of the lexicographically sorted suffixes of the word *mathematics*$ together the \( SA(mathematics\$) \) and the \( bwt(mathematics\$) \).
For instance, if $W = \text{mathematics}$ then $\text{bwt}(W$) = $\text{smmiht}$sceaa and $\text{SA}(W$) = $[12, 2, 7, 10, 5, 4, 9, 1, 6, 10, 2, 8]$. The table obtained by lexicographically sorting all the suffixes of $W$ is depicted in Figure [4].

3 Local and global suffixes of a text

Let $W \in \Sigma^*$ and let $W = L_1L_2\cdots L_k$ be its Lyndon Factorization. For each factor $L_r$, we denote by $\text{first}(L_r)$ and $\text{last}(L_r)$ the position of the first and the last character, respectively, of the factor $L_r$ in $W$. Let $u$ be a factor of $W$. We denote by $\text{ suf }_u(i) = W[i, \text{ last}(u)]$ and we call it local suffix at the position $i$ with respect to $u$. Note that $\text{ suf }_W(i) = W[i, n]$ and we call it global suffix of $W$ at the position $i$. We write $\text{ suf } (i)$ instead of $\text{ suf }_W(i)$ when there is no danger of ambiguity.

**Definition 2.** Let $W$ be a word and let $u$ be a factor of $W$. We say that the sorting of suffixes of $u$ is compatible with the sorting of suffixes of $W$ if for all $i, j$ with $\text{first}(u) \leq i < j \leq \text{last}(u)$,

$$\text{ suf }_u(i) < \text{ suf }_u(j) \iff \text{ suf } (i) < \text{ suf } (j).$$

Notice that in general taken an arbitrary factor of a word $W$, the sorting of its suffixes is not compatible with the sorting of the suffixes of $W$. Consider for instance the word $W = \text{abababb}$ and its factor $u = \text{ababa}$. Then $\text{ suf }_u(1) = \text{ababa} > a = \text{ suf }_u(5)$ whereas $\text{ suf } (1) = \text{abababb} < abb = \text{ suf } (5)$.

**Theorem 3.** Let $W \in \Sigma^*$ and let $W = L_1L_2\cdots L_k$ be its Lyndon factorization. Let $u = L_sL_{s+1}\cdots L_v$. Then the sorting of the suffixes of $u$ is compatible with the sorting of the suffixes of $W$.

Proof. Let $i$ and $j$ be two indexes with $i < j$ both contained in $u$. We just need to prove that $\text{ suf } (i) > \text{ suf } (j) \iff \text{ suf }_u(i) > \text{ suf }_u(j)$. Let $x = W[i, \text{ last}(L_s)]$ and $y = W[i, i+|x|-1]$.

Suppose that $\text{ suf } (i) > \text{ suf } (j)$. Then $y \geq x$ by the definition of lexicographic order. If $y > x$ there is nothing to prove. If $x = y$, then $\text{ suf }_u(j)$ is prefix of $\text{ suf }_u(i)$, so by the definition of lexicographic order $\text{ suf }_u(j) > \text{ suf }_u(i)$.

Suppose now that $\text{ suf }_u(i) > \text{ suf }_u(j)$. This means that $y \geq x$. If $y > x$ there is nothing to prove. If $x = y$, the index $i+|x|-1$ is in some Lyndon factor $L_m$ with $r \leq m \leq s$, then $L_r \geq L_m \geq L_s$. We denote $z = W[i+|x|, \text{ last}(L_m)]$. Then $\text{ suf } (i) = xzL_{m+1}\cdots L_k > xL_{s+1}\cdots L_k = \text{ suf } (j)$, since $z > L_m$ (because $L_m$ is a Lyndon word) and $L_m \geq L_{s+1}$ (since the factorization is a sequence of non increasing factors).

The above theorem states, in other words, that mutual order of the suffixes of $W$ starting in two positions $i$ and $j$ is the same as the mutual order of the “local” suffixes starting in $i$ and $j$ inside the block obtained as concatenation of the consecutive Lyndon factors including $i$ and $j$.

As particular case, the theorem is also true when the two suffixes start in the same Lyndon factor.

We recall that, if $l_1$ and $l_2$ denote two sorted lists of elements taken from any well ordered set, the operation $\text{merge}(l_1, l_2)$ consists in obtaining the sorted list of elements in $l_1$ and $l_2$.

A consequence of previous theorem is stated in the following proposition.
Proposition 4. Let \( \text{sort}(L_1L_2 \cdots L_i) \) and \( \text{sort}(L_{i+1}L_{i+2} \cdots L_k) \) denote the sorted lists of the suffixes of \( L_1L_2 \cdots L_i \) and the suffixes \( L_{i+1}L_{i+2} \cdots L_k \), respectively. Then \( \text{sort}(L_1L_2 \cdots L_k) = \text{merge}(\text{sort}(L_1L_2 \cdots L_i), \text{sort}(L_{i+1}L_{i+2} \cdots L_k)) \).

This proposition suggests a possible strategy for sorting the list of the suffixes of some word \( W \):

- find the Lyndon decomposition of \( W \), \( L_1L_2 \cdots L_k \);
- find the sorted list of the suffixes of \( L_1 \) and, separately, the sorted list of the suffixes of \( L_2 \);
- merge the sorted lists in order to obtain the sorted lists of the suffixes of \( L_1L_2 \);
- find the sorted list of the suffixes of \( L_3 \) and merge it to the previous sorted list;
- keep on this way until all the Lyndon factors are processed;

This kind of strategy could have several advantages: first of all, one can work online, i.e. one can start sorting suffixes as soon as the first Lyndon factor is individuated. This also allow to integrate the sorting process with the Duval’s Algorithm for Lyndon decomposition that outputs Lyndon factors online as well.

The second advantage is that this kind of strategy allows parallelization, since every Lyndon factor can be processed separately for sorting its suffixes. These kind of application would require an efficient algorithm to perform the merging of two sorted lists.

A detailed algorithmic description of this method in order to obtain the \( \text{bwt} \) of a text is given in next section.

4 An incremental algorithm to sort suffixes of a text

In this section we propose an algorithm that incrementally constructs the suffix array \( \text{SA} \) and the Burrows-Wheeler transform \( \text{bwt} \) of the text \( W \) by using its Lyndon factorization. In particular, here we detail the construction of the \( \text{bwt} \) but an analogous reasoning can be done in order to obtain the suffix array. We assume that \( L_1L_2 \cdots L_k \) is the Lyndon factorization of the word \( W[1,n] \). So \( L_1 \geq L_2 \geq \cdots \geq L_k \). Such an hypothesis, although strong, is not restrictive because one can obtain the Lyndon factorization of any word in linear time (cf. [11,18]). As shown in previous section, the hypothesis that \( W \) is factorized in Lyndon words suggests to connect the problem to the sorting of the local suffixes of \( W \) to the lexicographic sorting of the global suffixes of \( W \).

Our algorithm, called \( \text{BWT-LYND} \), considers the input text \( W[1,n] \) as logically partitioned into \( k \) blocks, where each block corresponds to a Lyndon word, and computes incrementally the \( \text{bwt}(W\$) \) via \( k \) iterations, one per block of \( W \). Each block is examined from right to left so that at iteration \( i \) we compute \( \text{bwt}(L_1 \cdots L_i\$) \) given \( \text{bwt}(L_1 \cdots L_{i-1}\$) \), \( \text{bwt}(L_i\$) \) and \( \text{SA}(L_i\$) \). Remark that the positions in \( \text{SA}(L_1\$) \) range in \([\text{first}(L_i), \text{Last}(L_i) + 1]\). This means that we sum the amount \(|L_1 \cdots L_{i-1}|\) to the values of the usual suffix array of \( L_i\$ \).

The key point of the algorithm comes from Theorem 3 because the construction of \( \text{bwt}(L_1 \cdots L_i\$) \) from \( \text{bwt}(L_1 \cdots L_{i-1}\$) \) requires only the insertion of the characters of \( L_i \) in \( \text{bwt}(L_1 \cdots L_{i-1}\$) \) in the same mutual order as they appear in \( \text{bwt}(L_i\$) \). Note that the character \( \$ \) that follows \( L_i \) is not considered in this operation.

Moreover, such an operation does not modify the mutual order of the characters already lying in \( \text{bwt}(L_1 \cdots L_{i-1}\$) \).
For each block $L_i$ with $i$ ranging from 1 to $k$, the algorithm BWT\_LYND executes the following steps:

1. Compute the bwt($L_i$) and SA($L_i$).
2. Compute the counter array $G[1, |L_i| + 1]$ which stores in $G[j]$ the number of suffixes of the string $L_1 \cdots L_{i-1}$ which are lexicographically smaller than the $j$-th suffix of $L_i$.
3. Merge bwt($L_1 \cdots L_{i-1}$) and bwt($L_i$) in order to obtain bwt($L_1 \cdots L_{i-1} L_i$).

Example 5. Let $W = aabcabbaabaabdbaaabbbdc$. The Lyndon factorization of $W$ is $L_1 L_2 L_3$, where $L_1 = aabcab > L_2 = aabaabdbb > L_3 = aaabdbdc$. Figure 2 illustrates how Step 3 of the algorithm works. Note that in the algorithm BWT\_LYND we do not actually compute the sorted list of suffixes, but we show it in Figure 2 to ease the comprehension of the algorithm. Moreover, the algorithm can be simply adapt to compute the suffix array of $W$, so in Figure 2 the suffix arrays are also shown.

![Figure 2](image-url)
suffixes of the string $L_1 \cdots L_{i-1}$ which are lexicographically smaller than the suffix of $L_i$ starting at the position $j$. Remark that we set $A[1] = 0$ because $L_i[1], [L_i]_S$ has the same rank of $S$ between the suffixes of $L_1 \cdots L_{i-1}$ and it is preceded by the same symbol $L_{i-1}(\lfloor L_{i-1} \rfloor)$ in $L_1 \cdots L_{i-1}$. Consequently, in our algorithm considers the suffixes $L_i[1], [L_i]_S$ and the suffix $S$ (of the string $L_1 \cdots L_{i-1}$) as the same suffix. It is easy to prove that the value $A[\lfloor L_i \rfloor + 1]$ is 0. The array $A$ is computed from the position $\lfloor L_i \rfloor$ to 2 by using Proposition 3.

**Proposition 6.** Let $j$ be a integer ranging from $|L_i|$ to 2 and let $A[j + 1]$ be the number of suffixes of $L_1 \cdots L_{i-1}$ lexicographically smaller than $L_i[j + 1], [L_i]_S$. Let $c$ be the first symbol of the suffix $L_i[j, |L_i|]$. Then,

$$A[j] = C(bwt(L_1 \cdots L_{i-1}), c) + \text{rank}(bwt(L_1 \cdots L_{i-1}), c, A[j + 1]).$$

**Proof.** Since c is the first symbol of the suffix $L_i[j, |L_i|]$, then $L_i[j, |L_i|] = cL_i[j + 1, |L_i|]$. All the suffixes of $L_1 \cdots L_{i-1}$ starting with a symbol smaller than $c$ are lexicographically smaller than $L_i[j, |L_i|]$. The number of such suffixes is given by $C(bwt(L_1 \cdots L_{i-1}), c)$. Let us count now the number of suffixes that starting with $c$ and are smaller than $L_i[j, |L_i|]$. This is equivalent to counting how many $c$'s occur in $bwt(L_1 \cdots L_{i-1})[1, A[j + 1]]$. A such value is given by $\text{rank}(bwt(L_1 \cdots L_{i-1}), c, A[j + 1])$. $\square$

It is easy to verify that we can obtain the array $G$ by using the array $A$ and the suffix array $SA(L_i)$, i.e., $G[i] = A[SA(L_i)[i]]$. Note that the array $G$ contains the partial sums of the values of the gap array used in $SA$. However, we could directly compute the array $G$ by using the notion of inverse suffix array $ISA$1. Step 2 could be realized in $O(\sum_{j=1,\ldots,|L_j|})$ time because we can build a data structure supporting $O(1)$ time rank queries over $bwt(L_1 \cdots L_{i-1})$. The same time complexity is obtained if the rank queries are executed over $bwt(L_i)$.

Step 3 uses $G$ to create the new array $bwt(L_1 \cdots L_i)$ by merging $bwt(L_i)$ with the $bwt(L_1 \cdots L_{i-1})$ computed at the previous iteration. Such a step implicitly constructs the lexicographically sorted list of suffixes starting in $L_1 \cdots L_{i-1}$ and extending up to the end of $L_i$ together with the suffixes of $L_i$. In order to do this we keep the mutual order between the suffixes of $L_1 \cdots L_{i-1}$ and $L_i$ thanks to Theorem 3. From the definition of the array $G$, it follows that the first two positions of the array $bwt(L_1 \cdots L_i)$ are the first symbol of $bwt(L_i)$ and the first symbol of $bwt(L_1 \cdots L_{i-1})$, respectively. For $j = 3, \ldots, |L_i|$ we copy $G[j]$ values from $bwt(L_1 \cdots L_{i-1})$ followed by the value $bwt(L_i)[j]$. It is easy to see that the time complexity of Step 3 is $O(\sum_{j=1,\ldots,|L_j|}$, too.

From the description of the algorithm and by proceeding by induction, one can prove the following proposition.

**Proposition 7.** At the end of the iteration $k$, Algorithm BWT_\text{LYND} correctly computes $bwt(L_1 \cdots L_k)$. Each iteration $i$ runs in $O(\sum_{j=1,\ldots,i} |L_j|)$ time. The overall time complexity is $O(k^2M)$, where $M = \max_{i=1,\ldots,k}(|L_i|)$.

1 The inverse suffix array $ISA$ of a word $W$ is the inverse permutation of $SA$, i.e., $ISA[SA[i]] = i$ for all $i \in [1, |w| + 1]$. The value $ISA[j]$ is the lexicographical rank of the suffix starting at the position $j$. 
5 Discussions and conclusions

The goal of this paper is to propose a new strategy to compute the \textit{bwt} and the \textit{SA} of a text by decomposing it into Lyndon factors and by using the compatibility relation between the sorting of its local and global suffixes. At the moment, the quadratic cost of the algorithm could make it impractical. However, from one hand, in order to improve our algorithm, efficient dynamic data structure for the rank operations and for the insertion operations could be used. Navarro and Nekrich’s recent result \cite{21} on optimal representations of dynamic sequences shows that one can insert symbols at arbitrary positions and compute the rank function in the optimal time $O\left(\frac{\log n}{\log \log n}\right)$ within essentially $nH_0(s) + O(n)$ bits of space, for a sequence $s$ of length $n$. On the other hand, our technique, differently from other approaches in which partitions of the text are performed, is quite versatile so that it easily can be adapted to different implementative scenarios.

For instance, in \cite{12} the authors describe an algorithm, called \textit{bwte}, that logically partitions the input text $W$ of length $n$ into blocks of the same length $m$, i.e. $W = T_{n/m}T_{n/m-1} \cdots T_1$ and computes incrementally the \textit{bwt} of $W$ via $n/m$ iterations, one per block of $W$. Text blocks are examined from right to left so that at iteration $h+1$, they compute and store on disk \textit{bwt}(\textit{T}_{h+1} \cdots \textit{T}_1) given \textit{bwt}(\textit{T}_h \cdots \textit{T}_1).$ In this case the mutual order of the suffixes in each block depends on the order of the suffixes of the next block. Our algorithm \textit{Bwt$_\text{Lynd}$} builds the \textit{bwt} of a text or its \textit{SA} by scanning the text \textit{from left to right} and it could run online, i.e. while the Lyndon factorization is realized. One of the advantages is that adding new text to the end does not imply to compute again the mutual order of the suffixes of the text analyzed before, unless for the suffixes of the last Lyndon word that could change by adding characters on the right. Moreover, as described in the previous section, the text could be partitioned into several sequences of consecutive blocks of Lyndon words, and the algorithm can be applied \textit{in parallel} to each of those sequences. Furthermore, also the Lyndon factorization can be performed in parallel, as shown in \cite{3}. Alternatively, since we read each symbol only once, also an in-place computation could be suggested by the strategy proposed in \cite{11}, in which the space occupied by text $W$ is used to store the \textit{bwt}(\textit{W}).

Finally, in the description of the algorithm we did not mention the used workspace. In fact, it could depend on the time-space trade-off that one should reach. For instance, the methodologies used in \cite{4,12} where disk data access are executed only via sequential scans could be adapted in order to obtain a lightweight version of the algorithm. An external memory algorithm for the Lyndon factorization can be found in \cite{24}. We remark that that the method proposed in \cite{12} could be integrated into \textit{Bwt$_\text{Lynd}$} in the sense that one can apply \textit{bwte} to compute at each iteration the \textit{bwt} and the \textit{SA} of each block of the Lyndon partition.

In conclusion, our method seems lay out the path towards a new approach to the problem of sorting the suffixes of a text in which partitioning the text by using its combinatorial properties allows it to tackle the problem in local portions of the text in order to extend efficiently solutions to a global dimension.

References