



On Inverting the Burrows-Wheeler Transform

Nicola Cotumaccio  

University of Helsinki, Finland

Abstract

We study the relationship between four fundamental problems: sorting, suffix sorting, element distinctness and BWT inversion. Our main contribution is an $\Omega(n \log n)$ lower bound for BWT inversion in the comparison model. As a corollary, we obtain a new proof of the classical $\Omega(n \log n)$ lower bound for sorting, which we believe to be of didactic interest for those who are not familiar with the Burrows-Wheeler transform.

2012 ACM Subject Classification Theory of computation \rightarrow Design and analysis of algorithms; Theory of computation \rightarrow Models of computation

Keywords and phrases Burrows-Wheeler transform, sorting, suffix array, element distinctness

Digital Object Identifier 10.4230/OASICS.Grossi.2025.17

Category Research

Funding Funded by the Helsinki Institute for Information Technology (HIIT).

1 Introduction

Sorting is typically one of the introductory topics in a first course on algorithms and data structures. Knuth devotes almost four hundred pages to the problem in the *The Art of Computer Programming* [12] and Cormen et al. use insertion sort as a first example of an algorithm [3].

Consider a sorted alphabet (i.e., an alphabet endowed with a total order). The problem of sorting can be stated as follows.

► **Problem 1 (Sorting).** *Given a string $T = a_1 a_2 \dots a_n$, compute a permutation ψ of $\{1, 2, \dots, n\}$ such that $a_{\psi(i)} \leq a_{\psi(i+1)}$ for every $1 \leq i \leq n - 1$.*

For example, if $T = cdba$, then the output of Problem 1 is the permutation ψ of $\{1, 2, 3, 4\}$ such that $\psi(1) = 4$, $\psi(2) = 3$, $\psi(3) = 1$ and $\psi(4) = 2$. Note that the permutation ψ of Problem 1 is uniquely defined if and only the a_i 's are pairwise distinct. More generally, ψ becomes uniquely defined if we add the additional requirement that, for every $1 \leq i \leq n - 1$, if $a_{\psi(i)} = a_{\psi(i+1)}$, then $\psi(i) < \psi(i + 1)$. The permutation ψ that satisfies this additional requirement is the *stable sort* of T .

Both Knuth and Cormen et al.'s consider the *comparison model*, in which no restriction on the (possibly infinite) sorted alphabet is assumed, and the only way to obtain information on the mutual order between the characters in T is by solving queries $c(i, j)$ of the following type: given $1 \leq i, j \leq n$, decide whether $a_i \leq a_j$. We assume that each query $c(i, j)$ takes $O(1)$ time.

In the comparison model, the (worst-case) complexity of Problem 1 is $\Theta(n \log n)$: there exist algorithms (e.g., merge sort, which computes the stable sort of T) solving Problem 1 in $O(n \log n)$ time, and any algorithm solving Problem 1 has complexity $\Omega(n \log n)$. The (classical) proof of the $\Omega(n \log n)$ lower bound [12, 3] shows a stronger result: to solve Problem 1, in the worst case we need $\Omega(n \log n)$ queries $c(i, j)$'s, and this is true even if we know that the a_i 's are pairwise distinct. In other words, $\Omega(n \log n)$ is not only an algorithmic lower bound, but it also captures the minimum number of operations required to have enough information for solving Problem 1.



© Nicola Cotumaccio;

licensed under Creative Commons License CC-BY 4.0

From Strings to Graphs, and Back Again: A Festschrift for Roberto Grossi's 60th Birthday.

Editors: Alessio Conte, Andrea Marino, Giovanna Rosone, and Jeffrey Scott Vitter; Article No. 17; pp. 17:1–17:8

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The comparison model yields a simple, informative, and mathematically appealing setting for studying the complexity of sorting, but it can hardly be considered a realistic computational model. For example, if the a_i 's are known to be integers in a polynomial range, Problem 1 can be solved in $O(n)$ time via radix sort, which also computes the stable sort of T [10].

The landscape becomes more complex if one considers alternative models of computations. On the one hand, it is possible to obtain models that are more flexible than the comparison model by allowing more complex queries in $O(1)$ time. Assume that the alphabet is the set of all real numbers, and consider the function $f(x, y) = x - y$. In the comparison model, in $O(1)$ time we can test whether $f(x, y) \leq 0$ for any choice of $x, y \in \{a_1, a_2, \dots, a_n\}$. In the *linear decision tree model*, one is allowed more general tests of the type $f(x_1, x_2, \dots, x_n) \leq 0$, where we consider a linear function $f(x_1, x_2, \dots, x_n) = c_0 + c_1x_1 + c_2x_2 + \dots + c_nx_n$. In the *algebraic decision tree model*, we can choose $f(x_1, x_2, \dots, x_n)$ to be an arbitrary polynomial. Even if the linear decision tree model and the algebraic decision tree model are more general than the comparison model, the complexity of Problem 1 in all these models is still $\Theta(n \log n)$ [4, 1]. On the other hand, it is possible to consider (realistic) variants of the RAM model. In the word-RAM model with word size $w \geq \log N$, where N is the size of the input, the complexity of Problem 1 is still open, even though it is known to be $o(n \log n)$ [8, 9]. Other variants of these models are possible, but we will focus only on the comparison model and the case of integers in a polynomial range, which are arguably the most common settings in the literature.

In this paper, we show that the complexity of Problem 1 (Sorting) is the same as the complexity of other fundamental problems: suffix sorting, element distinctness and BWT inversion. The complexity of these problems is $\Theta(n \log n)$ in the comparison model and $\Theta(n)$ in the case of integers in a polynomial range. Our main contribution is an $\Omega(n \log n)$ lower bound for BWT inversion in the comparison model (Theorem 1). We also show that Theorem 1 implies a new proof of the classical $\Omega(n \log n)$ lower bound for sorting (Corollary 2), which we believe to be of didactic interest for those who are not familiar with the Burrows-Wheeler transform.

2 Sorting, BWT Inversion and Related Problems

Problem 1 is closely related to several fundamental problems. Here are some examples.

► **Problem 2** (Suffix sorting). *Given a string $T = a_1a_2 \dots a_n$, compute the permutation ψ of $\{1, 2, \dots, n\}$ such that $a_{\psi(i)}a_{\psi(i)+1} \dots a_{n-1}a_n$ is the i -th lexicographically smallest suffix of T for every $1 \leq i \leq n$.*

For example, if $T = \text{banana}$, then the output of Problem 2 is the permutation ψ of $\{1, 2, 3, 4, 5, 6\}$ such that $\psi(1) = 6$, $\psi(2) = 4$, $\psi(3) = 2$, $\psi(4) = 1$, $\psi(5) = 5$ and $\psi(6) = 3$. Note that ψ is uniquely defined because the suffixes of a string are pairwise distinct.

► **Problem 3** (Element distinctness). *Given a string $T = a_1a_2 \dots a_n$, decide whether there exist $1 \leq i < j \leq n$ such that $a_i = a_j$.*

For example, if $T = \text{cdba}$, then the output of Problem 3 is “no”.

The complexity of Problems 2 and 3 is $\Theta(n)$ in the case of integers in a polynomial range, and $\Theta(n \log n)$ in the comparison model. Let us show how to obtain these bounds by reducing these problems to Problem 1 (Sorting).

Let us consider the case of integers in a polynomial range.

- Problem 2 can be solved in $O(n)$ time by using any linear-time algorithm for building the suffix array (see [15] for a survey). Historically, the linear time complexity was first proved by Farach, who solved the more general problem of building the suffix tree of a string in linear time [5]. Conceptually, the easiest algorithm is probably Kärkkäinen et al.'s algorithm [11].
- Problem 3 can be solved in $O(n)$ time by first computing a permutation ψ as in Problem 1 (Sorting) and then checking in $O(n)$ time if there exists $1 \leq i \leq n-1$ such that $a_{\psi(i)} = a_{\psi(i+1)}$.

Let us consider the comparison model.

- On the one hand, Problem 2 can be solved in $O(n \log n)$ time by (i) sorting the a_i 's via any $O(n \log n)$ comparison-based algorithm, (ii) replacing each a_i with its rank in the sorted list of all a_i 's (which does not affect the mutual order of the suffixes) and (iii) applying any $O(n)$ suffix sorting algorithm mentioned earlier. On the other hand, to solve Problem 2 we need $\Omega(n \log n)$ queries $c(i, j)$'s in the worst case, and the same lower bound is true even if we know that the a_i 's are pairwise distinct. Indeed, given the permutation ψ of Problem 2, we have $a_{\psi(i)} \leq a_{\psi(i+1)}$ for every $1 \leq i \leq n-1$ (because the suffix $a_{\psi(i)}a_{\psi(i)+1} \dots a_{n-1}a_n$ is lexicographically smaller than the suffix $a_{\psi(i+1)}a_{\psi(i+1)+1} \dots a_{n-1}a_n$), so ψ is also a correct output for Problem 1, and the conclusion follows from the lower bound for Problem 1 mentioned earlier.
- On the one hand, Problem 3 has complexity $O(n \log n)$: we can argue as in the case of integers in a polynomial range, but we use any $O(n \log n)$ comparison-based algorithm to compute a permutation ψ as in Problem 1. On the other hand, to solve Problem 3 we need $\Omega(n \log n)$ queries $c(i, j)$'s in the worst case, as we show next. Fix an integer n , and consider an algorithm that solves Problem 3 for every string $T = a_1a_2 \dots a_n$. Let $f(n)$ be the number of queries $c(i, j)$'s solved by the algorithm in the worst case. If the a_i 's are pairwise distinct, the permutation ψ of Problem 1 is uniquely defined, and the algorithm for Problem 3 must return “no”. To this end, the algorithm must solve the query $c(\psi(i+1), \psi(i))$ for every $1 \leq i \leq n-1$ because otherwise the algorithm could not infer that $a_{\psi(i)} \neq a_{\psi(i+1)}$ from the other queries that it solves and so it would not have enough information to solve Problem 3 correctly on input T . Consequently, if for every query $c(i, j)$ solved by the algorithm we also consider the query $c(j, i)$, we obtain at most $2f(n)$ queries. In particular, we consider both the query $c(\psi(i+1), \psi(i))$ and the query $c(\psi(i), \psi(i+1))$ for every $1 \leq i \leq n-1$, from which we can infer that $a_{\psi(i)} < a_{\psi(i+1)}$ for every $1 \leq i \leq n-1$. We conclude that at most $2f(n)$ queries $c(i, j)$'s are sufficient in the worst case to compute ψ and so solve Problem 1 for every $T = a_1a_2 \dots a_n$ in which the a_i 's are pairwise distinct. Hence, we obtain $f(n) = \Omega(n \log n)$ from the lower bound for Problem 1 mentioned earlier.

Let us show that Problem 1 is related to another fundamental problem in string processing and compression. To this end, we need to introduce the *Burrows-Wheeler transform (BWT)* of a string [2]. Let $S = b_1b_2 \dots b_n$ be a string such that $b_n = \$$, where $\$$ is a special character such that (i) $\$$ does not occur anywhere else in the string and (ii) $\$$ is smaller than all the other characters. For example, we can consider the string $S = \text{banana}\$$ (where $n = 7$). For $1 \leq i \leq n$, let $S_i = b_ib_{i+1} \dots b_nb_1b_2 \dots b_{i-1}$ be the i -th *circular suffix* of S . For example, if $S = \text{banana}\$$, we have $S_1 = \text{banana}\$, S_2 = \text{anana}\$, S_3 = \text{nana}\$, S_4 = \text{ana}\$, S_5 = \text{na}\$, S_6 = \text{a}\$, S_7 = \text{\$banana}$. Note that the circular suffixes of S are pairwise distinct because $\$$ occurs in a different position in each of them.

17:4 On Inverting the Burrows-Wheeler Transform

■ **Table 1** The Burrows-Wheeler transform of *banana\$*.

i	$F[i]$						$L[i]$	ψ	A
1	\$	<i>b</i>	<i>a</i>	<i>n</i>	<i>a</i>	<i>n</i>	<i>a</i>	5	7
2	<i>a</i>	\$	<i>b</i>	<i>a</i>	<i>n</i>	<i>a</i>	<i>n</i>	1	6
3	<i>a</i>	<i>n</i>	<i>a</i>	\$	<i>b</i>	<i>a</i>	<i>n</i>	6	4
4	<i>a</i>	<i>n</i>	<i>a</i>	<i>n</i>	<i>a</i>	\$	<i>b</i>	7	2
5	<i>b</i>	<i>a</i>	<i>n</i>	<i>a</i>	<i>n</i>	<i>a</i>	\$	4	1
6	<i>n</i>	<i>a</i>	\$	<i>b</i>	<i>a</i>	<i>n</i>	<i>a</i>	2	5
7	<i>n</i>	<i>a</i>	<i>n</i>	<i>a</i>	\$	<i>b</i>	<i>a</i>	3	3

We build the square matrix M of size $n \times n$ such that, for every $1 \leq i \leq n$, the i -th row $R_i[1, n]$ is equal to i -th (lexicographically) smallest circular suffix of S (see Table 1 for the matrix M of size 7×7 obtained from $S = \textit{banana\$}$). Note that $R_1 = b_n b_1 b_2 \dots b_{n-1} = S_n$ because $b_n = \$$ is the smallest character.

For $1 \leq i \leq n$, let $C_i[1, n]$ be the i -th column of M from left to right. Notice that every C_i is a rearrangement of the characters in S . Let $F = C_1$ and $L = C_n$ be the first column and the last column of M , respectively. In Table 1, we have $F = \$aaabnn$ and $L = annb\$aa$. Note that F can be obtained by sorting the characters of S . By definition, the Burrows-Wheeler transform $\text{BWT}[S]$ of S is the column F , that is, $\text{BWT}[S] = F = C_n$. In Table 1, we have $\text{BWT}[S] = annb\$aa$.

Crucially, the Burrows-Wheeler transform $\text{BWT}[S]$ of a string S is an *encoding* of the string: given $\text{BWT}[S]$, we can retrieve the string S . In Table 1, given $\text{BWT}[S] = annb\$aa$, we can retrieve $S = \textit{banana\$}$. To prove this, we only need to show that from $\text{BWT}[S]$ we can retrieve the matrix M , because then the unique row of the matrix ending with $\$$ yields the original string S . We can retrieve M by computing all columns C_i 's. We know that $C_n = \text{BWT}(S)$, and we can retrieve C_1 by sorting all characters in C_n . Let us show how to retrieve C_2 . We know C_n and C_1 , so we know all pairs of consecutive characters in S . If we sort these pairs lexicographically and we pick the last element of each pair, we retrieve C_2 . In Table 1, all pairs of consecutive characters in S are $a\$$, na , na , ba , $\$b$, an , an . By sorting these pairs, we obtain $\$b$, $a\$$, an , an , ba , na , na , and by picking the last element of each pair, we can infer that $C_2 = b\$nnaaa$. Let us show how to retrieve C_3 . We know C_n , C_1 and C_2 , so we know all triples of consecutive characters in S . If we sort these triples lexicographically and we pick the last element of each triple, we retrieve C_3 . In Table 1, all triples of consecutive characters in S are $a\$b$, $na\$$, nan , ban , $\$ba$, ana , ana . By sorting these triples, we obtain $\$ba$, $a\$b$, ana , ana , ban , $na\$$, nan , and by picking the last element of each triple, we can infer that $C_3 = abaan\$n$. In the same way, we can retrieve C_4, C_5, \dots, C_{n-1} .

Since $\text{BWT}(S)$ is an encoding of S , we can store $\text{BWT}(S)$ instead of S without losing information. The reason why storing $\text{BWT}(S)$ may be more beneficial than storing S is that the string $\text{BWT}(S)$ tends to be repetitive if in S several substrings have multiple occurrences, so we can exploit the repetitiveness of S to *compress* $\text{BWT}(S)$. This property motivated the introduction of the Burrows-Wheeler transform in the original paper [2] and can be stated in precise mathematical terms via the notion of *entropy* [13]. Surprisingly, it is also possible to solve pattern matching *on the original string* by augmenting the compressed representation of the Burrows-Wheeler transform with space-efficient data structures, thus obtaining the *FM-index* [6].

We described an (inefficient) algorithm to *invert* the Burrows-Wheeler transform. Let us state the problem formally.

► **Problem 4** (BWT inversion). *Given a string $T = a_1a_2 \dots a_n$ such that $T = \text{BWT}(S)$ for some ($\$$ -terminated) string S , compute S .*

For example, if $T = annb\$aa$, then $S = banana\$$ (see Table 1).

3 Our Results

In this section, we show that the complexity of Problem 4 is $\Theta(n \log n)$ in the comparison model and $\Theta(n)$ in the case of integers in a polynomial range. The only bound that cannot be inferred from the original paper on the Burrows-Wheeler transform [2] is the $\Omega(n \log n)$ lower bound in the comparison model. Notice that the four problems considered in this paper have the same complexity.

To prove the $\Omega(n \log n)$ bound for Problem 4, we will proceed differently from Problems 2 and 3. We will prove the lower bound directly, without relying on the lower bound for Problem 1 (Sorting). Then, we will use the lower bound for Problem 4 (BWT inversion) to infer the lower bound for Problem 1 (Sorting). In addition to establishing interesting relationships between fundamental problems, our approach yields a new proof of the celebrated $\Omega(n \log n)$ bound for sorting, which we believe to be of didactic interest.

Let us start with the lower bound for BWT inversion.

► **Theorem 1.** *In the comparison model, to solve Problem 4 we need $\Omega(n \log n)$ queries $c(i, j)$'s in the worst case. The same lower bound holds even if we know that the a_i 's are pairwise distinct.*

Proof. Fix an integer n . Consider $n - 1$ distinct characters b_1, b_2, \dots, b_{n-1} such that $\$ < b_1 < b_2 < \dots < b_{n-1}$. Then, the set:

$$\mathcal{S} = \{b_{\phi(1)}b_{\phi(2)} \dots b_{\phi(n-1)}\$ \mid \phi \text{ is a permutation of } \{1, 2, \dots, n-1\}\}$$

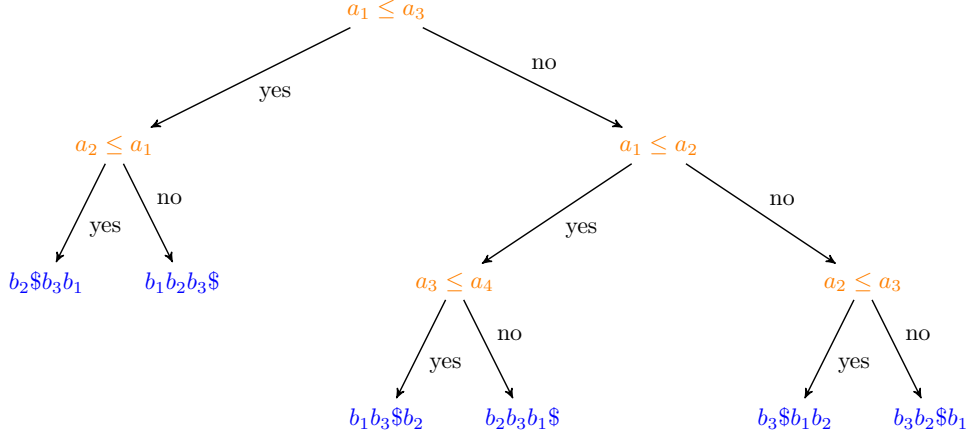
has size $(n - 1)!$. For every $S \in \mathcal{S}$, the string $\text{BWT}(S)$ is an encoding of S , so the set:

$$\mathcal{T} = \{\text{BWT}(S) \mid S \in \mathcal{S}\}$$

has also size $(n - 1)!$. Notice that for every string $T = a_1a_2 \dots a_n$, if $T \in \mathcal{T}$, then the a_i 's are pairwise distinct.

Consider *any* algorithm solving Problem 4 for every input $T = a_1a_2 \dots a_n \in \mathcal{T}$. The algorithm can gather information on the mutual order between the a_i 's only by solving queries $c(i, j)$'s. The decision on the next query $c(i, j)$ can depend on the outcome of the previous queries $c(i, j)$'s, so we can describe the behavior of the algorithm on all inputs $T \in \mathcal{T}$ through a *decision tree* (see Figure 1 for the case $n = 4$). Since the algorithm correctly solves Problem 4 for every input $T = a_1a_2 \dots a_n \in \mathcal{T}$, then the outcome of all queries $c(i, j)$'s on a path from the leaf to a root cannot be consistent with two distinct elements of \mathcal{T} , otherwise the algorithm would not have enough information to compute S . For example, in Figure 1, if the output of “ $a_1 \leq a_3$ ” is “no” and then the output “ $a_1 \leq a_2$ ” is “yes”, then the algorithm must necessarily solve an additional query $c(i, j)$: both $T_1 = b_1b_3\$b_2$ and $T_2 = b_2b_3b_1\$$ are strings in \mathcal{T} for which $\neg(a_1 \leq a_3) \wedge (a_1 \leq a_2)$, and we have $T_1 = \text{BWT}(S_1)$ and $T_2 = \text{BWT}(S_2)$ for two distinct $S_1, S_2 \in \mathcal{S}$, where $S_1 = b_2b_3b_1\$$ and $S_2 = b_3b_1b_2\$$.

Assume that the longest path in the tree consists of k edges. Then, the number of paths from the root to a leaf is upper bounded by 2^k , so we must have $2^k \geq (n - 1)!$, which implies $k = \Omega(n \log n)$. This means that there exists $T \in \mathcal{T}$ for which the algorithm needs to solve $\Omega(n \log n)$ queries $c(i, j)$'s. ◀



■ **Figure 1** The decision tree of a possible algorithm solving Problem 4 (see the proof of Theorem 1) for $n = 4$. We have $|S| = |\mathcal{T}| = (4 - 1)! = 6$. The tree describes the sequence of all queries $c(i, j)$'s for every input $T = a_1a_2a_3a_4 \in \mathcal{T}$. Recall that we assume $\$ < b_1 < b_2 < b_3$. We have $\text{BWT}(b_1b_2b_3\$) = b_3\b_1b_2 , $\text{BWT}(b_1b_3b_2\$) = b_2\b_3b_1 , $\text{BWT}(b_2b_1b_3\$) = b_3b_2\b_1 , $\text{BWT}(b_2b_3b_1\$) = b_1b_3\b_2 , $\text{BWT}(b_3b_1b_2\$) = b_2b_3b_1\$$ and $\text{BWT}(b_3b_2b_1\$) = b_1b_2b_3\$$. Every element of \mathcal{T} corresponds to a path from the root to a leaf.

The proof of Theorem 1 is similar to the classical proof of the $\Omega(n \log n)$ lower bound for sorting [12, 3]. In the classical proof, one typically uses the fact that a binary tree with $n!$ leaves must have height at least $\log(n!)$. Here we used a slightly more direct pigeonhole argument to prove the inequality $2^k \geq (n - 1)!$. The *worst-case entropy* of a set S is $\log |S|$ by a similar pigeonhole argument [14], so one may argue that in the proof of Theorem 1 we used an entropy-based argument. This appears to establish an interesting analogy because we have already mentioned that the compressibility of the Burrows-Wheeler transform can be described through the notion of entropy [13].

Now, let us describe an efficient algorithm to solve Problem 4. The original paper by Burrows and Wheeler [2] follows an approach based on a permutation called *LF-mapping*. Here we will use a different approach based on the *permutation* ψ , which is the inverse of the LF-mapping. The permutation ψ plays a crucial role in compressed suffix arrays [7], and it captures the connection between Problem 4 (BWT inversion) and Problem 1 (Sorting) more explicitly. We are given $T = a_1a_2 \dots a_n$ such that $T = \text{BWT}(S)$ for some $S = b_1b_1 \dots b_n$, where $b_n = \$$, and we need to compute S . By definition, $T = L$, where L is the last column of the matrix M , so we know that $L = a_1a_2 \dots a_n$ and we must retrieve S .

Let ψ be the stable sort of L (see Problem 1 and Table 1). Since $\$$ is the smallest character, we have $L[\psi(1)] = \$ = b_n$. We will prove that $b_i = L[\psi^{i+1}(1)]$ for every $1 \leq i \leq n - 1$. For example, in Table 1 we have $b_1 = L[\psi^2(1)] = L[4] = b$, $b_2 = L[\psi^3(1)] = L[7] = a$, $b_3 = L[\psi^4(1)] = L[3] = n$, $b_4 = L[\psi^5(1)] = L[6] = a$, $b_5 = L[\psi^6(1)] = L[2] = n$ and $b_6 = L[\psi^7(1)] = L[1] = a$, so $S = \text{banana}\$$. After computing ψ , we can retrieve all the b_i 's in $O(n)$ time by computing all the powers $\psi^{i+1}(1)$'s. In the comparison model, we can compute ψ in $O(n \log n)$ time, and in the case of integers in a polynomial range we can compute ψ in $O(n)$ time, so the complexity of Problem 4 is $O(n \log n)$ in the comparison model and $O(n)$ in the case of integers in a polynomial range. We are left with proving that $b_i = L[\psi^{i+1}(1)]$ for every $1 \leq i \leq n - 1$.

Let $A[1, n]$ be the array such that, for every $1 \leq i \leq n$, the row R_i of the matrix M is equal to the circular suffix $S_{A[i]}$ (see Table 1). Then, $A[1, n]$ yields a permutation of $\{1, 2, \dots, n\}$. Moreover, we have $F[i] = b_{A[i]}$ and $L[i] = b_{A[i]-1}$ for every $1 \leq i \leq n$, where we assume $b_0 = b_n$. In particular, $b_{A[\psi(1)]-1} = L[\psi(1)] = \$$, so we have $A[\psi(1)] = 1$ and $2 \leq A[\psi(i)] \leq n$ for $2 \leq i \leq n$. Notice that A yields a permutation of $\{1, 2, \dots, n\}$. From $R_1 = S_n$ we obtain $A[1] = n$.

Let us prove that $A[\psi(i)] = A[i] + 1$ for every $2 \leq i \leq n$ (for example, in Table 1 we have $A[\psi(2)] = A[1] = 7 = 6 + 1 = A[2] + 1$). Fix a character $c \neq \$$ that occurs in S . Let $1 \leq i_1 \leq n$ be the smallest integer such that $F[i_1] = c$, and let $1 \leq i_2 \leq n$ be the largest integer such that $F[i_2] = c$. We only have to prove that $A[\psi(i)] = A[i] + 1$ for every $i_1 \leq i \leq i_2$, because by picking all possible values $c \neq \$$ from smallest to largest we cover every i between 2 and n ($i = 1$ corresponds to $\$$). Let us prove that $A[\psi(i)] = A[i] + 1$ for every $i_1 \leq i \leq i_2$. From the definitions of i_1 and i_2 we obtain that in every row of M and in every column of M there are exactly $i_2 - i_1 + 1$ characters equal to c , and in particular $L[\psi(i_1)] = L[\psi(i_1 + 1)] = \dots = L[\psi(i_2 - 1)] = L[\psi(i_2)] = c$. Since ψ is the stable sort of L , we obtain $\psi(i_1) < \psi(i_1 + 1) < \dots < \psi(i_2 - 1) < \psi(i_2)$. This implies that $S_{A[\psi(i_1)]}$ is lexicographically smaller than $S_{A[\psi(i_1+1)]}$, $S_{A[\psi(i_1+1)]}$ is lexicographically smaller than $S_{A[\psi(i_1+2)]}$, \dots , $S_{A[\psi(i_2-1)]}$ is lexicographically smaller than $S_{A[\psi(i_2)]}$. We have $b_{A[\psi(i)]-1} = L[\psi(i)] = c$ for every $i_1 \leq i \leq i_2$, so we conclude that $S_{A[\psi(i_1)]-1}$ is lexicographically smaller than $S_{A[\psi(i_1+1)]-1}$, $S_{A[\psi(i_1+1)]-1}$ is lexicographically smaller than $S_{A[\psi(i_1+2)]-1}$, \dots , $S_{A[\psi(i_2-1)]-1}$ is lexicographically smaller than $S_{A[\psi(i_2)]-1}$, where $b_{A[\psi(i)]-1} = c$ for every $i_1 \leq i \leq i_2$. At the same time, for every $1 \leq i \leq n$ we have $b_{A[i]} = c$ if and only if $i_1 \leq i \leq i_2$, and $S_{A[i_1]}$ is lexicographically smaller than $S_{A[i_1+1]}$, $S_{A[i_1+1]}$ is lexicographically smaller than $S_{A[i_1+2]}$, \dots , $S_{A[i_2-1]}$ is lexicographically smaller than $S_{A[i_2]}$. We obtain $A[\psi(i)] - 1 = A[i]$ for every $i_1 \leq i \leq i_2$, so $A[\psi(i)] = A[i] + 1$ for every $i_1 \leq i \leq i_2$, as claimed.

Let us prove that $A[\psi^i(1)] = i$ for every $1 \leq i \leq n$ (for example, in Table 1 we have $A[\psi^2(1)] = A[\psi(5)] = A[4] = 2$). We proceed by induction on i . For $i = 1$, we know that $A[\psi(1)] = 1$. Now assume that $2 \leq i \leq n$. By the inductive hypothesis, we know that $A[\psi^{i-1}(1)] = i - 1$. In particular, $A[\psi^{i-1}(1)] \neq n$, so $2 \leq \psi^{i-1}(1) \leq n$ and we obtain $A[\psi^i(1)] = A[\psi(\psi^{i-1}(1))] = A[\psi^{i-1}(1)] + 1 = (i - 1) + 1 = i$.

We are now ready to prove the main claim. We have $b_i = b_{(i+1)-1} = b_{A[\psi^{i+1}(1)]-1} = L[\psi^{i+1}(1)]$ for every $1 \leq i \leq n - 1$.

We conclude our paper by showing that Theorem 1 implies a new proof of the lower bound for sorting.

► **Corollary 2.** *In the comparison model, to solve Problem 1 we need $\Omega(n \log n)$ queries $c(i, j)$'s in the worst case. The same lower bound holds even if we know that the a_i 's are pairwise distinct.*

Proof. Consider any algorithm solving Problem 4 for every input $T = a_1 a_2 \dots a_n$ such that the a_i 's are pairwise distinct. Since the a_i 's are pairwise distinct, the permutation ψ of Problem 1 is uniquely defined, and ψ is also the stable sort of T . Let $f(n)$ be the number of queries $c(i, j)$'s solved by the algorithm in the worst case to compute ψ . Assume that $T = \text{BWT}(S)$, where $S = b_1 b_2 \dots b_n$. After computing ψ , we can compute S in $O(n)$ time (because $b_i = T[\psi^{i+1}(1)]$ for every $1 \leq i \leq n - 1$, as we have seen before) without solving any additional query $c(i, j)$. Consequently, $f(n)$ queries are sufficient in the worst case to solve Problem 4 for every input $T = a_1 a_2 \dots a_n$ such that the a_i 's are pairwise distinct. By Theorem 1, we conclude $f(n) = \Omega(n \log n)$. ◀

4 Conclusions

In this paper, we have shown that, in the comparison model, inverting the Burrows-Wheeler transform has complexity $\Theta(n \log n)$. As a corollary, we have obtained a new proof of the $\Omega(n \log n)$ sorting lower bound. Our main goal was to highlight how the ideas behind the Burrows-Wheeler transform are deeply intertwined with the most fundamental results in Computer Science.

References

- 1 Michael Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the fifteenth Annual ACM Symposium on Theory of Computing*, pages 80–86, 1983.
- 2 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Systems Research Center, 1994.
- 3 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.
- 4 David P Dobkin and Richard J Lipton. On the complexity of computations under varying sets of primitives. *Journal of Computer and System Sciences*, 18(1):86–91, 1979. doi:10.1016/0022-0000(79)90054-0.
- 5 Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997. doi:10.1109/SFCS.1997.646102.
- 6 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000. doi:10.1109/SFCS.2000.892127.
- 7 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the thirty-second Annual ACM Symposium on Theory of Computing*, pages 397–406, 2000.
- 8 Torben Hagerup. Sorting and searching on the word RAM. In *STACS 98: 15th Annual Symposium on Theoretical Aspects of Computer Science Paris, France, February 25–27, 1998 Proceedings 15*, pages 366–398. Springer, 1998. doi:10.1007/BFB0028575.
- 9 Yijie Han. Deterministic sorting in $O(n \log \log n)$ time and linear space. *Journal of Algorithms*, 50(1):96–105, 2004. doi:10.1016/J.JALGOR.2003.09.001.
- 10 John E Hopcroft, Jeffrey D Ullman, and Alfred Vaino Aho. *Data structures and algorithms*, volume 175. Addison-wesley Boston, MA, USA:, 1983.
- 11 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.
- 12 Donald E Knuth. *The Art of Computer Programming: Sorting and Searching, volume 3*. Addison-Wesley Professional, 1998.
- 13 Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM (JACM)*, 48(3):407–430, 2001. doi:10.1145/382780.382782.
- 14 Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.
- 15 Simon J Puglisi, William F Smyth, and Andrew H Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys (CSUR)*, 39(2):4–es, 2007.