Constructing Suffixient Arrays Revisited

Paola Bonizzoni
Department of Computer Science, University of Milano-Bicocca, Italy

Younan Gao
Department of Computer Science, University of Milano-Bicocca, Italy

Brian Riccardi
Department of Computer Science, University of Milano-Bicocca, Italy

Abstract

Recently, Cenzato et al. proposed a new text index, called the *suffixient array*, which is a subset of the suffix array and supports locating a single pattern occurrence or finding its maximal exact matches (MEMs), assuming random access to the input text $T[1..n]$ is available. They show that, given the suffix array, the longest common prefix array, and the Burrows–Wheeler transform (`BWT`) of the reverse of $T[1..n]$ over an alphabet $\{1, \ldots, \sigma\}$, a suffixient array can be constructed in linear time. However, their construction algorithms require multiple scans of these arrays. When restricted to a single pass over the arrays, they present an alternative construction algorithm running in $\mathcal{O}(n + \overline{r} \log \sigma)$ time, where $\overline{r}$ is the number of runs in the `BWT` of the reversed text. In this paper, we present a new one-pass algorithm that constructs a suffixient array in linear time under the standard RAM model.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Suffixient set, suffixient array, right-maximal substring, linear-time algorithm

Digital Object Identifier 10.4230/LIPIcs.CPM.2026.30

Funding All authors have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement PANGAIA No. 872539, as well as from grant MIUR 2022YRB97K (PINC, *Pangenome Informatics: From Theory to Applications*), funded by the European Union under the NextGenerationEU programme, Mission 4.

1 Introduction

Pattern matching is a fundamental problem with applications ranging from text processing to computational biology. The *suffix array* [12] is a textbook data structure that supports efficient pattern matching by storing the starting positions of all suffixes of a text in lexicographical order. Together with auxiliary structures such as the *longest common prefix array* [12], the suffix array enables fast pattern searches and serves as the basis of many full-text indexes. However, the limitations of suffix arrays become apparent when dealing with massive texts, such as collections of human genomes. Since a suffix array requires linear space in the text length, storing and processing it for such datasets is often infeasible in practice.

At the same time, genomic data exhibits a high degree of similarity and repetitiveness: genomes from different individuals share the vast majority of their sequences, with variations occurring only at relatively sparse locations. This strong repetitiveness suggests that designing space-efficient indexes whose space usage scales with intrinsic measures of repetitiveness, rather than with the raw text length n, is both natural and highly desirable.

Suffixient arrays [5] were recently proposed as a space-efficient alternative to suffix arrays for indexing highly similar texts such as genome sequences. Intuitively, a suffixient array can be viewed as a carefully selected subset of the *prefix array*, the symmetric counterpart of the suffix array. Its definition is grounded in the notion of *right-maximal substrings*, that is, substrings that occur in the text and can be extended to the right by at least two distinct

characters. Such substrings correspond exactly to the internal nodes of the suffix tree. The suffixient array captures these branching points by including prefixes of the text that cover all one-character right-maximal extensions, ensuring that the structure is sufficient to represent all essential distinctions induced by right-maximal extensions.

Despite its reduced size, a suffixient array supports a specialized set of queries, provided random access to the underlying text: it can be used to either locate a single occurrence of a given pattern or to find *maximal exact matches* (MEMs), which are of particular importance in bioinformatics applications [5].

In this paper, we focus on the efficient construction of suffixient arrays. Unlike suffix arrays, which are uniquely determined for a given text, a text may admit multiple valid suffixient arrays. Our goal is to compute any one of them.

Related Work. Recently, Navarro et al. [13] introduced an *online construction algorithm* that computes a *minimum-size suffixient set* – the set of indices forming a suffixient array – directly from the text. Their approach adapts Ukkonen's algorithm [15], originally designed for the online construction of the suffix *tree*. Specifically, their method incrementally maintains a minimum-size suffixient set for the prefix $T[1..i]$ as the text is scanned left to right.

The algorithm runs in linear time over an alphabet of size σ under the trans-dichotomous RAM model [8], but requires $\mathcal{O}(n \log \sigma)$ time in the standard RAM model [2]. Its working space is comparable to that of a suffix tree built over the text, which is known to have a large memory footprint in practice – up to 20 bytes per input character in the worst case [10]. Navarro et al. also demonstrated that a minimum-size suffixient set can be constructed incrementally while scanning the text from right to left.

Cenzato et al. [5] addressed the problem of constructing a minimum-size suffixient set given the suffix array $\texttt{SA}[1..n]$, the Burrows–Wheeler transform (BWT) [4] array $\texttt{BWT}[1..n]$, and the longest common prefix array $\texttt{LCP}[1..n]$ of the reversed text. They presented two linear-time algorithms that require multiple passes over these arrays, resulting in a working space of $\Omega(n)$ words. They also proposed a construction algorithm that avoids storing the full SA, BWT, and LCP arrays and processes them in a single pass. This algorithm runs in $\mathcal{O}(n + \overline{r} \log \sigma)$ time, where $\overline{r}$ denotes the number of runs in the BWT, and uses only $\mathcal{O}(\sigma)$ words of working space. When computing a suffixient array, the running time remains unchanged, while the working space increases from $\mathcal{O}(\sigma)$ to $\mathcal{O}(\sigma + \chi) \subseteq \mathcal{O}(\chi)$ words, where χ denotes the size of the suffixient array and $\sigma \leq \chi$ [5].

Using the technique of *prefix-free parsing* (PFP) [3, 9, 14], the entries of the SA, BWT, and LCP arrays can be generated in a left-to-right streaming fashion while using compressed space. By combining their one-pass algorithm with PFP, Cenzato et al. [5] were able to construct a suffixient array in compressed space.

Our Results. In this paper, we revisit the problem of constructing a suffixient array in the standard RAM model, focusing on the specific setting introduced by Cenzato et al. [5]. Our main contribution is a linear-time streaming algorithm that constructs a suffixient array in n iterations by processing the SA, LCP, and BWT arrays of the reversed text $T[1..n]$ in a single pass with one-step look-ahead. The efficiency of the algorithm relies on two key mechanisms.

Dynamic Candidate Management. At each iteration i, we maintain a set of candidate positions j_{text} for the suffixient array in a doubly-linked list (rowList). The list has size at most σ and is kept sorted in non-increasing order by the weight $\texttt{w}(j)$. Here, $\texttt{w}(j)$ denotes the maximum length of a right-maximal substring that is a suffix of $T[1..j_{\text{text}} - 1]$, where $j = \texttt{SA}^{-1}[n - j_{\text{text}} + 1]$. The array $\texttt{SA}^{-1}$ denotes the inverse suffix array, defined by $\texttt{SA}^{-1}[\texttt{SA}[k]] = k$.

To decide at iteration i whether the new position $i_{\text{text}} = n - \texttt{SA}[i] + 1$ should replace or join the existing candidates, we use an auxiliary array $\texttt{prevW}[1..\sigma]$ that records, for each character, the most recent weighted occurrence (its *index*) together with the corresponding *weight*. In addition, we employ a monotone stack to compute in amortized $\mathcal{O}(1)$ time $\texttt{b}(i)$, the largest index $i' < i$ such that $\texttt{LCP}[i'] < \texttt{LCP}[i]$. By comparing $\texttt{w}(i)$ with $\texttt{LCP}[i]$, $\texttt{prevW}[c].\textit{index}$ with $\texttt{b}(i)$, and $\texttt{prevW}[c].\textit{weight}$ with $\texttt{w}(i)$ (where $c = \texttt{BWT}[i]$), we can determine whether the position i_{text} is a superior candidate to those currently stored. This strategy guarantees that, at any time, we retain only the most relevant candidate for each distinct character. ⌟

The Ejection Mechanism. As the algorithm streams through the arrays, the current LCP value, $\texttt{LCP}[i]$, acts as a threshold. When $\texttt{LCP}[i]$ drops below the weight of a candidate in rowList, this indicates that the corresponding right-maximal extension can no longer be extended. The candidate *position* is then *ejected* from the doubly-linked list and appended to the output list, which stores a sublist of the suffixient array. Since rowList is sorted, each ejection takes constant time, resulting in overall linear running time. ⌟

Our algorithm is asymptotically faster than the one-pass algorithm of Cenzato et al. [5]. Its working space is bounded by $\mathcal{O}(h + \chi)$, in addition to the SA, LCP, and BWT arrays, while computing a minimum-size suffixient set requires only $\mathcal{O}(h+\sigma)$ words of working space. Here, h denotes the height of the suffix tree built over the reversed text, that is, the maximum number of branching nodes along any root-to-leaf path. Since the algorithm processes these arrays in a one-pass streaming fashion, it naturally integrates with prefix-free parsing [3].

2 Preliminaries

Our model of computation is a random access machine (RAM) endowed with comparison operations and basic arithmetic operations, including only addition and subtraction [2].

Let $\Sigma = \{1, \ldots, \sigma\}$ be the alphabet set. We assume that the text $T[1..n]$ ends at the position n with a special character \$ that never appears in $T[1..n-1]$. So, the alphabet size is at least two. Without loss of generality, we further assume that $\sigma \leq n$.

For any string S, let $(S)^{rev}$ denote its reverse. We define the reversed text $T^{rev}[1..n]$ as a specific construction distinct from $(T)^{rev}$; while $(T)^{rev}$ simply reverses the entire sequence, T^{rev} preserves the sentinel \$ at the final position. Formally, $T^{rev}[n] = \$$ and $T^{rev}[i] = T[n-i]$ for $1 \leq i < n$. Let $\texttt{SA}[1..n]$, $\texttt{BWT}[1..n]$, and $\texttt{LCP}[1..n]$ denote the suffix array, the Burrows–Wheeler transform, and the Longest Common Prefix array of the reversed text T^{rev}, respectively. Specifically, $\texttt{SA}[1..n]$ is a permutation of the indices $\{1, \ldots, n\}$ such that the suffixes $T^{rev}[\texttt{SA}[i]..n]$ are arranged in lexicographical order for $1 \leq i \leq n$. And $\texttt{BWT}[i] = \$$ if $\texttt{SA}[i] = 1$; otherwise, $\texttt{BWT}[i] = T^{rev}[\texttt{SA}[i] - 1] = T[n - \texttt{SA}[i] + 1]$. Moreover, $\texttt{LCP}[1] = -1$, and for $i > 1$, $\texttt{LCP}[i]$ is the length of the longest common prefix of $(T[1..n - \texttt{SA}[i]])^{rev}$ and $(T[1..n - \texttt{SA}[i-1]])^{rev}$.

Given two strings α and β, we say that α is *co-lexicographically smaller* than β if and only if one of the following holds: i) there exists an index k such that $\alpha[|\alpha| - i + 1] = \beta[|\beta| - i + 1]$ for all $1 \leq i < k$, and $\alpha[|\alpha| - k + 1] < \beta[|\beta| - k + 1]$; ii) or α is a proper suffix of β.

Throughout this paper, we reserve the term *position* for locations in the text $T[1..n]$, and the term *index* for locations in the arrays SA, BWT, and LCP. For example, we use the symbol j_{text} to denote a position in $T[1..n]$, and write j (without the subscript "text") to denote an index in these arrays. Consequently, for $1 \leq j \leq n$ we have $j_{\text{text}} = n - \texttt{SA}[j] + 1$, and for $1 \leq j_{\text{text}} \leq n$ we have $j = \texttt{SA}^{-1}[\, n - j_{\text{text}} + 1\,]$.

▶ **Definition 1** (Right-maximal substrings and one-character right-maximal extension [5]). *For a text $T[1..n]$ containing at least two distinct characters, including \$, a substring $T[i_{text}..j_{text}]$ ($j_{text} \geq i_{text} - 1$) is a* right-maximal substring *if there exist at least two distinct characters $a, b \in \Sigma$ such that both $T[i_{text}..j_{text}] \cdot a$ and $T[i_{text}..j_{text}] \cdot b$ are substrings of T. For any right-maximal substring str, we call $str \cdot c$ for $c \in \Sigma$ a* one-character right-maximal extension *of str if $str \cdot c$ is a substring of $T[1..n]$.*

Note that the empty string is always a right-maximal substring, since $\sigma \geq 2$.

▶ **Definition 2** (Suffixient set [5]). *A set $\mathcal{X} \subseteq \{1, \ldots, n\}$ is* suffixient *for a text $T[1..n]$ if, for every one-character right-maximal extension $T[i_{text}..j_{text}]$ ($j_{text} \geq i_{text}$) of every right-maximal string $T[i_{text}..j_{text} - 1]$, there exists a position $x_{text} \in \mathcal{X}$ such that $T[i_{text}..j_{text}]$ is a suffix of $T[1..x_{text}]$.*

Since $T[n] = \$$ and \$ does not appear in $T[1..n-1]$, any suffixient set must contain n.

▶ **Definition 3** (Suffixient array [5]). *A* suffixient array *sA of a text $T[1..n]$ is a minimum-size suffixient set for T, whose elements are ordered so that the prefixes $T[1..sA[1]], T[1..sA[2]], \ldots$ appear in colexicographic order.*

Computing the $\mathtt{w}(\cdot)$ values. Let $x \in [1..n]$ be an index. We say that x is a *start-run boundary* if $x > 1$ and $\mathtt{BWT}[x] \neq \mathtt{BWT}[x-1]$, and an *end-run boundary* if $x < n$ and $\mathtt{BWT}[x] \neq \mathtt{BWT}[x+1]$. Any index that is either a start- or end-run boundary is called a *run boundary*.

To every index $x \in [1..n]$ we assign a *weight* $\mathtt{w}(x)$, defined as the maximum length of a right-maximal substring that is a suffix of $T[1..n - \mathtt{SA}[x]]$, if x is a run-boundary, and -1, otherwise. Note that if x is a run-boundary, then a right-maximal substring ending at position $n - \mathtt{SA}[x]$ surely exists, hence $\mathtt{w}(x)$ is well-defined.

The value $\mathtt{w}(x)$ can then be computed as follows: if x is not a run boundary, then we set $\mathtt{w}(x) := -1$, as per the definition. If x is only a start-run (resp., only an end-run) boundary, then $\mathtt{w}(x) = \mathtt{LCP}[x]$ (resp., $\mathtt{w}(x) = \mathtt{LCP}[x+1]$). If x is both a start- and end-run boundary, then $\mathtt{w}(x) = \max\{\mathtt{LCP}[x], \mathtt{LCP}[x+1]\}$. Each value can be computed in $\mathcal{O}(1)$ time using $\mathcal{O}(1)$ working space. The pseudocode is deferred to Appendix A.

Computing the $\mathtt{b}(\cdot)$ values. For any $x \in [1..n]$, define $\mathtt{b}(x)$ as the largest index $\mathtt{b}(x) < x$ for which $\mathtt{LCP}[\mathtt{b}(x)] < \mathtt{LCP}[x]$. If no such index exists, set $\mathtt{b}(x) := 1$. Define $\mathtt{e}(x)$ as the smallest index $\mathtt{e}(x) > x$ such that $\mathtt{LCP}[\mathtt{e}(x)] < \mathtt{LCP}[x]$. If no such index exists, set $\mathtt{e}(x) := n + 1$. The value $\mathtt{e}(x)$ is only used conceptually throughout this paper, so we only show an algorithm that computes $\mathtt{b}(x)$.

In our setting, the entries in $\mathtt{LCP}[1..n]$ are enumerated sequentially from $\mathtt{LCP}[1], \mathtt{LCP}[2], \ldots$ in a stream; immediately after reading the entry $\mathtt{LCP}[i]$ for each $i > 1$, our goal is to compute the index $\mathtt{b}(i)$ in amortized constant time.

Our data structure is a regular stack that stores tuples of form $(index, val, b\_val)$, where $index \in [1..n]$, $val = \mathtt{LCP}[index]$, and $b\_val = \mathtt{b}(index)$. We call this stack *monotone stack* as the tuples in the stack are always sorted decreasingly by the val entries from top to the bottom. Initially, we create an empty stack S and push the triple $(1, -1, -1)$ onto S.

When $\mathtt{LCP}[i]$ is available, we first check if or not the tuple at the top of S has $index = i$. If so, then we return the b_val value of the top tuple; this means that $\mathtt{b}(i)$ has already been computed before (note that in our setting, when $\mathtt{LCP}[i]$ is available, we might call $\mathtt{b}(i)$ more than once). Otherwise, we pop every tuple in the stack with $val \geq \mathtt{LCP}[i]$ out of S, until we find the tuple t with $t.val < \mathtt{LCP}[i]$. Then, we return $t.index$ right after we push the tuple $(i, \mathtt{LCP}[i], t.index)$ onto the top of S. The pseudocode can be found in Appendix B.

Since the tuple associated with each $\texttt{LCP}[i]$ is pushed onto S exactly once and popped from S at most once, the overall running time is $\mathcal{O}(n)$. Consequently, $\texttt{b}(i)$ can be computed in amortized constant time. The correctness is standard and omitted; a similar idea has appeared in [7, Theorem 2] and [1, Section 4] for a related purpose.

Let $|S|_i$ denote the size of the stack immediately after processing $\texttt{LCP}[i]$. By Proposition 4 (below), $\max\{|S|_i \mid 1 < i \leq n\}$ is upper bounded by the height of the *suffix tree* [16] built over $T^{rev}[1..n]$, plus one. In particular, if $T[1..n] = a^n$, then $|S| \subseteq \Omega(n)$, and if the text is uniform, then $|S| \subseteq \mathcal{O}(\log n)$ with high probability [6]. However, as noted in [1, Section 5.1], the stack size in practice is much smaller.

▶ **Proposition 4.** *During the sequential execution of* $\texttt{b}(i)$ *for* $i = 2, 3, \ldots, n$*, the size of the monotone stack is bounded by* $h+1$ *in the worst case, where* h *denotes the height of the suffix tree constructed over the reverse of the input text, that is the maximum number of branching nodes on any root–to–leaf path.*

Proof. Let $Trie$ denote the suffix tree constructed over $T^{rev}[1..n]$. For each i, let A_i denote the list of all index entries, excluding 1, of the triples stored in the monotone stack S (in increasing order) upon completion of the execution of $\texttt{b}(i)$. Thus, $|A_i| = |S|_i - 1$.

Consider any $j', j'' \in A_i$ with $j' < j''$, and any $s \in (j'..j'']$. Following the algorithm, we have $\texttt{LCP}[j'] < \texttt{LCP}[s]$; otherwise, the triple indexed by j' would have been popped from the stack. Consequently, $T^{rev}[\texttt{SA}[j']..\texttt{SA}[j'] + \texttt{LCP}[j'] - 1]$ is a prefix of $T^{rev}[\texttt{SA}[j'']..\texttt{SA}[j''] + \texttt{LCP}[j''] - 1]$.

For each $j \in A_i$, let u_j denote the node in $Trie$ that is the lowest common ancestor of the $\texttt{SA}[j-1]$-leaf and the $\texttt{SA}[j]$-leaf. Observe that u_j is the locus of the string $T^{rev}[\texttt{SA}[j]..\texttt{SA}[j] + \texttt{LCP}[j] - 1]$. Hence, $u_{j'}$ is an ancestor of $u_{j''}$, and the nodes $\{u_j \mid j \in A_i\}$ all lie on a single root-to-leaf path in $Trie$.

It follows that $|A_i| \leq h$. Since $|A_i| = |S|_i - 1$, we conclude that $|S|_i \leq h + 1$. ◀

3 A Smallest Suffixient Set: A New Characterization and Proof

In this section, we characterize a minimum-size suffixient set based the concepts of $\texttt{w}(\cdot), \texttt{b}(\cdot)$, and $\texttt{e}(\cdot)$ and prove its correctness.

▶ **Definition 5** (Candt). *Let* $1 \leq a < a' \leq n$ *and* $c \in \Sigma$*. Define* $\texttt{Candt}_c(a, a')$ *as the smallest index* $p \in [a..a')$ *with* $\texttt{BWT}[p] = c$ *and* $\texttt{w}(p) = \max\{\texttt{w}(x) \mid a \leq x < a',\ \texttt{BWT}[x] = c,\ \texttt{w}(x) \geq 0\}$*; if no such index exists, set* $\texttt{Candt}_c(a, a') := -1$*.*

By Definition 5, for any index x with $\texttt{BWT}[x] \neq \texttt{BWT}[x-1]$ and any $c \in \{\texttt{BWT}[x], \texttt{BWT}[x-1]\}$, it follows that $\texttt{Candt}_c(\texttt{b}(x), \texttt{e}(x)) \neq -1$. Indeed, both x and $x-1$ lie within the interval $[\texttt{b}(x), \texttt{e}(x))$, and $\texttt{w}(x-1) \geq 0$ as well as $\texttt{w}(x) \geq 0$. Observation 6 will be used throughout the remainder of the paper.

▶ **Observation 6.** *For any index* $1 < x \leq n$ *such that* $\texttt{BWT}[x] \neq \texttt{BWT}[x-1]$*, and any* $c \in \{\texttt{BWT}[x], \texttt{BWT}[x-1]\}$*, we have* $\texttt{w}(\ell_{x,c}) \geq \texttt{LCP}[x]$*, where* $\ell_{x,c} = \texttt{Candt}_c(\texttt{b}(x), \texttt{e}(x))$*.*

Proof. Since $\texttt{BWT}[x] \neq \texttt{BWT}[x-1]$ and $c \in \{\texttt{BWT}[x], \texttt{BWT}[x-1]\}$, Definition 5 implies that $\ell_{x,c} = \texttt{Candt}_c(\texttt{b}(x), \texttt{e}(x)) > -1$ and that $\texttt{w}(\ell_{x,c}) = \max\{\texttt{w}(j) \mid \texttt{b}(x) \leq j < \texttt{e}(x),\ \texttt{BWT}[j] = c\}$.

Let $\ell \in \{x, x-1\}$ be the index with $\texttt{BWT}[\ell] = c$. By the definition of $\texttt{w}(\cdot)$, it follows that $\texttt{w}(\ell) \geq \texttt{LCP}[x]$. Moreover, since $\ell \in [\texttt{b}(x), \texttt{e}(x))$ by the definitions of $\texttt{b}(\cdot)$ and $\texttt{e}(\cdot)$, and $\texttt{BWT}[\ell] = c$, the maximality of $\texttt{w}(\ell_{x,c})$ implies $\texttt{w}(\ell_{x,c}) \geq \texttt{w}(\ell) \geq \texttt{LCP}[x]$. ◀

Our goal is to show that $\texttt{FullL}$, defined in Definition 7, is a minimum-size suffixient set, thereby providing a new characterization of such sets in terms of $\mathtt{w}(\cdot)$, $\mathtt{b}(\cdot)$, and $\mathtt{e}(\cdot)$. To this end, one possible approach is to establish a bijection between $\texttt{FullL}$ and the set of all *super-maximal extensions*, as defined in [5, Definition 32]. This approach implies, as demonstrated in [5, Section 5], that $\texttt{FullL}$ is a minimum-size suffixient set. Instead, we present a new proof that does not rely on the notion of super-maximal extensions.

▶ **Definition 7** ($\texttt{FullL}$). *Define* $\texttt{FullL} = \{n - \texttt{SA}[p] + 1 \mid p = \texttt{Candt}_c(\mathtt{b}(x), \mathtt{e}(x)), 1 < x \leq n, \texttt{BWT}[x] \neq \texttt{BWT}[x-1], c \in \{\texttt{BWT}[x], \texttt{BWT}[x-1]\}\}$.

We first prove that $\texttt{FullL}$ is suffixient for the text $T[1..n]$, applying Definition 2.

▶ **Lemma 8.** *The set* $\texttt{FullL}$ *is suffixient for the text* $T[1..n]$.

Proof. By Definition 2, it suffices to show the following: for any right-maximal substring str of $T[1..n]$ and for any one-character right-maximal extension $str \cdot c$ of str, there exists a position $x_{\text{text}} \in \texttt{FullL}$ such that $str \cdot c$ is a suffix of $T[1..x_{\text{text}}]$.

Since str is right-maximal in $T[1..n]$, there exists another character $a \neq c$ such that $str \cdot a$ also occurs as a substring of $T[1..n]$. Consequently, there exists at least one interval $[\ell, z]$ such that both characters a and c occur in $\texttt{BWT}[\ell..z]$, and the prefix $T[1..n - \texttt{SA}[i]]$ has str as a suffix for every index $i \in [\ell, z]$.

Let $[\ell, z]$ be the smallest such interval. Without loss of generality, assume that $\texttt{BWT}[\ell] \neq c$ and $\texttt{BWT}[z] = c$; the case $\texttt{BWT}[\ell] = c$ and $\texttt{BWT}[z] \neq c$ is symmetric. By the minimality of the interval $[\ell, z]$, we have $\texttt{BWT}[z] \neq \texttt{BWT}[z-1]$. Let $q := \texttt{Candt}_c(\mathtt{b}(z), \mathtt{e}(z))$. Since $\texttt{BWT}[z] = c = \texttt{BWT}[q]$ and $\texttt{BWT}[z] \neq \texttt{BWT}[z-1]$, it follows that $\mathtt{w}(q) \geq \mathtt{w}(z) \geq \texttt{LCP}[z] \geq |str|$; moreover, as $T[n - \texttt{SA}[q] + 1] = \texttt{BWT}[q] = c$, we have $str \cdot c$ is a suffix of $T[1..n - \texttt{SA}[q] + 1]$. Setting $x_{\text{text}} := n - \texttt{SA}[q] + 1$, we conclude that $x_{\text{text}} \in \texttt{FullL}$, completing the proof. ◀

▶ **Lemma 9.** *Define* $\ell_{x,c} := \texttt{Candt}_c(\mathtt{b}(x), \mathtt{e}(x))$ *and* $\ell_{x',c'} := \texttt{Candt}_{c'}(\mathtt{b}(x'), \mathtt{e}(x'))$ *for any two start-run boundaries* x *and* x'*, and for any* $c \in \{\texttt{BWT}[x], \texttt{BWT}[x-1]\}$ *and* $c' \in \{\texttt{BWT}[x'], \texttt{BWT}[x'-1]\}$*, respectively. If* $\ell_{x,c} \neq \ell_{x',c'}$ *and* $\mathtt{w}(\ell_{x,c}) \leq \mathtt{w}(\ell_{x',c'})$*, then* $T[n - \texttt{SA}[\ell_{x,c}] + 1 - \mathtt{w}(\ell_{x,c})..\, n - \texttt{SA}[\ell_{x,c}] + 1]$ *cannot be a suffix of* $T[n - \texttt{SA}[\ell_{x',c'}] + 1 - \mathtt{w}(\ell_{x',c'})..\, n - \texttt{SA}[\ell_{x',c'}] + 1]$.

Proof. By Definition 5, we have $\ell_{x,c} > -1$, $\mathtt{w}(\ell_{x,c}) > -1$, $\ell_{x',c'} > -1$, and $\mathtt{w}(\ell_{x',c'}) > -1$.

Any index ℓ with $\mathtt{w}(\ell) \geq 0$ induces the right-maximal substring $T[n - \texttt{SA}[\ell] + 1 - \mathtt{w}(\ell),\, n - \texttt{SA}[\ell]]$, which cannot contain the symbol \$. Recall that the suffix array $\texttt{SA}$ is constructed over $T^{rev}[1..n]$. Therefore, the reverse of a right-maximal substring induces an interval in the suffix array, called its *SA-interval* [11, Section 2.2.2]. Formally, for any index ℓ with $\mathtt{w}(\ell) \geq 0$, let $[s(\ell), t(\ell)]$ denote the SA-interval of the string $T[n - \texttt{SA}[\ell] + 1 - \mathtt{w}(\ell)..\, n - \texttt{SA}[\ell]]$, where $s(\ell)$ is the minimum index $i \in [1..n]$ such that the string is a suffix of $T[1..n - \texttt{SA}[i]]$, and $t(\ell)$ is defined analogously as the maximum such index.

▷ Claim 10. We have $[s(\ell_{x,c}), t(\ell_{x,c})] \subseteq [\mathtt{b}(x), \mathtt{e}(x))$ and $[s(\ell_{x',c'}), t(\ell_{x',c'})] \subseteq [\mathtt{b}(x'), \mathtt{e}(x'))$.

Proof. It suffices to show that $s(\ell_{x,c}) \geq \mathtt{b}(x)$ and $t(\ell_{x,c}) < \mathtt{e}(x)$. By the definition of SA-intervals, $s(\ell_{x,c}) = \max\{i \in [1..\ell_{x,c}) \mid \texttt{LCP}[i] < \mathtt{w}(\ell_{x,c})\}$ and $t(\ell_{x,c}) = \min\{i \in [\ell_{x,c}..n] \mid \texttt{LCP}[i] < \mathtt{w}(\ell_{x,c})\} - 1$.

Assume for contradiction that $s(\ell_{x,c}) < \mathtt{b}(x)$. Since $\mathtt{b}(x) \leq \ell_{x,c}$, this implies $\texttt{LCP}[\mathtt{b}(x)] \geq \mathtt{w}(\ell_{x,c})$; otherwise, $s(\ell_{x,c}) \geq \mathtt{b}(x)$ by definition. By the definition of $\mathtt{b}(\cdot)$, $\texttt{LCP}[\mathtt{b}(x)] < \texttt{LCP}[x]$, yielding $\texttt{LCP}[x] > \mathtt{w}(\ell_{x,c})$, which contradicts Observation 6.

A symmetric argument shows that $t(\ell_{x,c}) < \mathtt{e}(x)$. If $\mathtt{e}(x) \leq t(\ell_{x,c})$, then since $\ell_{x,c} < \mathtt{e}(x)$ we obtain $\texttt{LCP}[\mathtt{e}(x)] \geq \mathtt{w}(\ell_{x,c})$, which together with $\texttt{LCP}[x] > \texttt{LCP}[\mathtt{e}(x)]$ again contradicts Observation 6. This proves the claim. ◁

We now prove the lemma by contradiction. Assume that $T[n - \mathtt{SA}[\ell_{x,c}] + 1 - \mathtt{w}(\ell_{x,c}) .. n - \mathtt{SA}[\ell_{x,c}] + 1]$ is a suffix of $T[n - \mathtt{SA}[\ell_{x',c'}] + 1 - \mathtt{w}(\ell_{x',c'}) .. n - \mathtt{SA}[\ell_{x',c'}] + 1]$, which implies $c = c'$. Consequently, $T[n - \mathtt{SA}[\ell_{x,c}] + 1 - \mathtt{w}(\ell_{x,c}) .. n - \mathtt{SA}[\ell_{x,c}]]$ is a suffix of $T[n - \mathtt{SA}[\ell_{x',c'}] + 1 - \mathtt{w}(\ell_{x',c'}) .. n - \mathtt{SA}[\ell_{x',c'}]]$, and hence $\ell_{x',c'} \in [s(\ell_{x,c}), t(\ell_{x,c})]$.

If $\mathtt{w}(\ell_{x,c}) < \mathtt{w}(\ell_{x',c'})$, then by Claim 10 we have $\ell_{x',c'} \in [\mathtt{b}(x), \mathtt{e}(x))$, which implies $\mathtt{w}(\mathtt{Candt}_c(\mathtt{b}(x), \mathtt{e}(x))) > \mathtt{w}(\ell_{x,c})$, contradicting the definition of $\ell_{x,c}$. If instead $\mathtt{w}(\ell_{x,c}) = \mathtt{w}(\ell_{x',c'})$, then the two substrings are equal, so $s(\ell_{x,c}) = s(\ell_{x',c'})$ and $t(\ell_{x,c}) = t(\ell_{x',c'})$. Assuming without loss of generality that $\ell_{x,c} < \ell_{x',c'}$, we obtain $\mathtt{b}(x') \le s(\ell_{x',c'}) \le \ell_{x,c} < \ell_{x',c'} \le t(\ell_{x',c'}) < \mathtt{e}(x')$, which contradicts the choice of $\ell_{x',c'}$ as $\mathtt{Candt}_{c'}(\mathtt{b}(x'), \mathtt{e}(x'))$. In both cases, we reach a contradiction, completing the proof. ◀

Finally, we show that $\mathtt{FullL}$ has minimum possible size, applying the *pigeonhole principle*.

▶ **Lemma 11.** *The list* $\mathtt{FullL}$ *is a minimum-size suffixient set.*

Proof. Let F be an arbitrary suffixient set. Our goal is to show that $|F| \ge |\mathtt{FullL}|$.

Suppose, for the sake of contradiction, that $|F| < |\mathtt{FullL}|$. Recall that each $x_{\text{text}} \in \mathtt{FullL}$ corresponds to a right-maximal extension $T[x_{\text{text}} - \mathtt{w}(\mathtt{SA}^{-1}[n - x_{\text{text}} + 1]) .. x_{\text{text}}]$. By the definition of suffixient sets, the string $T[x_{\text{text}} - \mathtt{w}(\mathtt{SA}^{-1}[n - x_{\text{text}} + 1]) .. x_{\text{text}}]$ must be a suffix of $T[1..f_{\text{text}}]$ for some $f_{\text{text}} \in F$.

Since $|F| < |\mathtt{FullL}|$, the pigeonhole principle implies that there exist two distinct indices $x_{\text{text}}, y_{\text{text}} \in \mathtt{FullL}$ such that both $T[x_{\text{text}} - \mathtt{w}(\mathtt{SA}^{-1}[n - x_{\text{text}} + 1]) .. x_{\text{text}}]$ and $T[y_{\text{text}} - \mathtt{w}(\mathtt{SA}^{-1}[n - y_{\text{text}} + 1]) .. y_{\text{text}}]$ are suffixes of the same prefix $T[1..f_{\text{text}}]$ for some $f_{\text{text}} \in F$. Consequently, one of these two strings must be a suffix of the other.

However, by Lemma 9, no such pair x_{text} and y_{text} can exist. This contradiction shows that our assumption was false, and therefore $|F| \ge |\mathtt{FullL}|$. ◀

Lemmas 11 and 8 together imply that $\mathtt{FullL}$ is a minimum-size suffixient set.

4 Computing sA: A One-Pass Algorithm with One-Step Look-ahead

In this section, we present a new one-pass algorithm that computes suffixient arrays. The algorithm iterates over the index i from 1 to n and makes decisions upon reading the triple $(\mathtt{BWT}[i], \mathtt{SA}[i], \mathtt{LCP}[i])$. During the iteration at index i, the algorithm may additionally access the input of the next iteration, namely the triple $(\mathtt{BWT}[i+1], \mathtt{SA}[i+1], \mathtt{LCP}[i+1])$, but it requires no information about inputs beyond $i + 1$. For this reason, we refer to it as a *one-pass algorithm with one-step look-ahead*.

An overview of this section is as follows. Section 4.1 introduces the data structures and the three invariants maintained by the algorithm. Section 4.2 then presents a detailed description of the algorithm together with the underlying intuitions. Since the algorithm invokes the operation $\mathtt{Candt}$ at each iteration, it may initially appear to require quadratic time. Before addressing the complexity analysis, we establish the correctness of the algorithm in Section 4.3. Next, Section 4.4 shows that a *validity test* for $\mathtt{Candt}$, such as checking whether $i = \mathtt{Candt}(\cdot, \cdot)$, can be implemented using a constant number of amortized $\mathcal{O}(1)$-time operations, thereby justifying that the algorithm runs in a one-pass fashion. Finally, Section 4.5 presents the overall complexity analysis.

4.1 The Data Structures

During the execution of the algorithm, we maintain the following data structures:

- A monotone stack used to compute $\mathtt{b}(i)$ during the i-th iteration of the algorithm;
- A doubly-linked list, `rowList`, containing at most σ triples drawn from $[1..n] \times \Sigma \times [0..n]$;
- An array $\mathtt{MAP}[1..\sigma]$ such that, for each $c \in \Sigma$, the entry $\mathtt{MAP}[c]$ stores a pointer to the (unique) triple $(\cdot, c, \cdot)$ in `rowList`, if such a triple exists, and stores `null` otherwise;
- For every $c \in \Sigma$, a linked list $\mathtt{result}_c$ containing positions drawn from $[1..n]$.

For convenience, we define `result` as the list obtained by concatenating all $\mathtt{result}_1, \ldots,$ $\mathtt{result}_\sigma$ in this order.

For every triple $(p_{\text{text}}, \cdot, \cdot)$ stored in `rowList`, we will refer to p_{text} as a *candidate position*, and to the corresponding index $\mathtt{SA}^{-1}[n - p_{\text{text}} + 1]$ as a *candidate index*.

The following invariants describe the content of the above data-structures during the i-th iteration of the algorithm. Among these, **Invariant 2** characterizes the candidate indices stored in `rowList`.

- **Invariant 1:** For each $c \in \Sigma$, $\mathtt{result}_c$ is the sub-list of `FullL` containing exactly those positions j_{text} such that $\mathtt{SA}^{-1}[n - j_{\text{text}} + 1] < i$ and $T[j_{\text{text}}] = c$.
- **Invariant 2:** The list `rowList` contains exactly those triples $(n - \mathtt{SA}[j] + 1, \mathtt{BWT}[j], \mathtt{w}(j))$ where the index j satisfies either of the following conditions:
  - $\mathtt{w}(j) = \mathtt{LCP}[j]$, $j \leq i - 1 < \mathtt{e}(j)$, and $\mathtt{Candt}_{\mathtt{BWT}[j]}(\mathtt{b}(j), i) = j$; or
  - $\mathtt{w}(j) > -1$, $\mathtt{w}(j) \neq \mathtt{LCP}[j]$, $j \leq i - 1 < \mathtt{e}(j+1)$, and $\mathtt{Candt}_{\mathtt{BWT}[j]}(\mathtt{b}(j+1), i) = j$.
- **Invariant 3:** The triples in `rowList` are sorted in non-increasing order by weight.

4.2 The Algorithm

In this section we give an high-level description of the algorithm. As already mentioned above, the algorithm iterates over indexes $i = 1, 2, \ldots, n$ once. In Step 0 we describe the special case of $i = 1$. Then, Step 1 and Step 2 are executed in every successive iteration $i = 2, 3 \ldots, n$. Finally, a last step will return the list `FullL` using the information computed in the preceding iterations.

For both Step 1 and 2 we give some intuitive explanation on why they are correct. The full proof is given in Section 4.3. The pseudocode of the algorithm is deferred to the Appendix C.

Step 0. In the first iteration, if $\mathtt{BWT}[2] \neq \mathtt{BWT}[1]$ we append the triple $(n - \mathtt{SA}[1] + 1, \mathtt{BWT}[1], \mathtt{w}(1))$ to the front of `rowList`, and we set $\mathtt{MAP}[\mathtt{BWT}[1]]$ to point to such triple; otherwise, no triple is inserted in `rowList`. Every list $\mathtt{result}_c$ remains empty.

We now consider iteration i for $i \in (1, n]$. The operations are divided into two steps.

Step 1. Remove from `rowList` every triple (p_{text}, c, w) such that $w > \mathtt{LCP}[i]$. Set $\mathtt{MAP}[c]$ to `null` and append the position p_{text} to the tail of the list $\mathtt{result}_c$.

Intuition behind Step 1. We provide some intuition for why the position p_{text} appended to $\mathtt{result}_{char}$ always belongs to `FullL`, a property required to maintain **Invariant 1**.

Let $j = \mathtt{SA}^{-1}[n - p_{\text{text}} + 1]$. By Lemma 15, depending on whether $\mathtt{LCP}[j] = w$, we have either $j = \mathtt{Candt}_c(\mathtt{b}(j), i)$ or $j = \mathtt{Candt}_c(\mathtt{b}(j+1), i)$. Moreover, Lemma 16 shows that when $\mathtt{LCP}[i] < w$, the corresponding interval endpoint satisfies $\mathtt{e}(j) = i$ in the former case and $\mathtt{e}(j+1) = i$ in the latter. Thus, intuitively, j is always selected as a candidate index over one of the intervals $[\mathtt{b}(j), \mathtt{e}(j))$ or $[\mathtt{b}(j+1), \mathtt{e}(j+1))$.

Furthermore, Lemma 17 guarantees that in the first case $\mathtt{BWT}[j] \neq \mathtt{BWT}[j-1]$, while in the second case $\mathtt{BWT}[j+1] \neq \mathtt{BWT}[j]$. In both situations, j satisfies the defining conditions of `FullL`. Therefore, the position p_{text} appended to $\mathtt{result}_c$ indeed belongs to `FullL`.

Lemma 13 states that the triples in rowList are sorted in non-increasing order of their weight values. As a result, every triple satisfying $\texttt{LCP}[i] < w$ can be identified and removed in $\mathcal{O}(1)$ time.

Step 2. Compute the value $\texttt{w}(i)$. If $\texttt{w}(i) = -1$, proceed immediately to the next iteration. Otherwise, compute the index $\texttt{b}(i)$ and initialize a variable curr_b to $\texttt{b}(i)$. If $\texttt{w}(i) \neq \texttt{LCP}[i]$, update curr_b to $\texttt{b}(i+1)$.

Assume that $c = \texttt{BWT}[i]$. We then check whether $i = \texttt{Candt}_c(\texttt{curr\_b}, i+1)$. If this condition does not hold, proceed immediately to the next iteration. If $\texttt{MAP}[c] \neq \texttt{null}$, remove from rowList the triple pointed to by $\texttt{MAP}[c]$. Then prepend the triple $(n - \texttt{SA}[i] + 1, c, \texttt{w}(i))$ to rowList and update $\texttt{MAP}[c]$ to point to the new head.

Intuition behind Step 2. We now give an informal explanation of why both **Invariant 2** and **Invariant 3** are maintained after this step. If either $\texttt{w}(i) = -1$ or $i \neq \texttt{Candt}_c(\texttt{curr\_b}, i+1)$, the algorithm immediately proceeds to the next iteration. In this situation, no triple is added to rowList, and both invariants are trivially maintained. Otherwise, the triple $(n - \texttt{SA}[i] + 1, \texttt{BWT}[i], \texttt{w}(i))$ is added to the front of rowList. By Lemma 13, this insertion preserves the non-increasing order of weights in rowList, ensuring that **Invariant 3** holds.

It remains to argue that the newly added triple satisfies one of the conditions in **Invariant 2**. Two cases arise, depending on whether $\texttt{LCP}[i] = \texttt{w}(i)$. If $\texttt{LCP}[i] = \texttt{w}(i)$, then $\texttt{curr\_b} = \texttt{b}(i)$ and $i = \texttt{Candt}_c(\texttt{b}(i), i+1)$; otherwise, $\texttt{LCP}[i] \neq \texttt{w}(i)$, $\texttt{curr\_b} = \texttt{b}(i+1)$, and $i = \texttt{Candt}_c(\texttt{b}(i+1), i+1)$. Setting $j := i$, we have $\texttt{w}(j) > -1$, and either $j \leq i+1-1 < \texttt{e}(j)$ with $\texttt{Candt}_c(\texttt{b}(j), i+1) = j$, or $j \leq i+1-1 < \texttt{e}(j+1)$ with $\texttt{Candt}_c(\texttt{b}(j+1), i+1) = j$. In both cases, the conditions of **Invariant 2** are satisfied. In Section 4.4, we show how the test $i = \texttt{Candt}_c(\texttt{curr\_b}, i+1)$ can be implemented in amortized constant time.

Final step after the n iterations. If rowList is nonempty, append all positions p_{text} from the remaining tuples (p_{text}, c, w) to their respective $\texttt{result}_c$ lists. Then all the $\texttt{result}_c$ lists are concatenated to obtain result, which is then returned as the suffixient array.

In the next section we prove that result = FullL. Hence, by Lemma 11, the output list is suffixient and of minimum size. Note that the prefix list $\{ T[1..\texttt{SA}[1]], \ldots, T[1..\texttt{SA}[n]] \}$ is sorted in co-lexicographic order. We further show that this ordering ensures that the indices in the output list are correctly sorted, in accordance with the definition of a suffixient array.

4.3 Correctness of the Algorithm

In this section, we prove the correctness of the algorithm. The proof is divided into three parts. In Section 4.3.1 we prove that the triples in rowList are always sorted non-increasingly by their weight values. Then, in Section 4.3.2, we show that for any character $c \in \Sigma$ and any position $\ell_{\text{text}} \in \texttt{result}_c$, it holds that $\ell_{\text{text}} \in \texttt{FullL}$. Finally, in Section 4.3.3, we prove the converse: every $\ell_{\text{text}} \in \texttt{FullL}$ is added to $\texttt{result}_c$ for some $c \in \Sigma$; we also show that the order of indices in the final list result is consistent with the order induced by the suffixient array.

4.3.1 The Ordering on the Triples in rowList

▶ **Lemma 12.** *Let $c = \texttt{BWT}[i]$. If $i = \texttt{Candt}_c(\texttt{curr\_b}, i+1)$ at iteration i and $\texttt{w}(i) \leq \texttt{LCP}[i]$, then one of the following holds: either $\texttt{BWT}[j] = c$ for every $j \in [\texttt{curr\_b}, i)$, or $\texttt{BWT}[j] \neq c$ for every $j \in [\texttt{curr\_b}, i)$.*

Proof. Suppose, for the sake of contradiction, that there exists an index $j' \in [\texttt{curr\_b}, i-1)$ such that either $\texttt{BWT}[j'] = c$ and $\texttt{BWT}[j'+1] \neq c$, or $\texttt{BWT}[j'] \neq c$ and $\texttt{BWT}[j'+1] = c$. Let $j'' \in \{j', j'+1\}$ be such that $\texttt{BWT}[j''] = c$.

We claim that $\texttt{w}(j'') \geq \texttt{LCP}[j'+1]$. Indeed, if $\texttt{BWT}[j'] = c$ and $\texttt{BWT}[j'+1] \neq c$, then by the definition of $\texttt{w}(\cdot)$ we have $\texttt{w}(j'') = \texttt{w}(j') \geq \texttt{LCP}[j'+1]$. Otherwise, if $\texttt{BWT}[j'] \neq c$ and $\texttt{BWT}[j'+1] = c$, then $\texttt{w}(j'') = \texttt{w}(j'+1) \geq \texttt{LCP}[j'+1]$.

Define $\ell := i$ if $\texttt{w}(i) = \texttt{LCP}[i]$, and otherwise $\ell := i+1$. By construction, $\texttt{b}(\ell) = \texttt{curr\_b}$ and $\texttt{w}(i) = \texttt{LCP}[\ell]$. Since $j'+1 \in (\texttt{curr\_b}, i)$ and $(\texttt{curr\_b}, i) \subseteq (\texttt{b}(\ell), \ell)$, the definition of $\texttt{b}(\cdot)$ implies $\texttt{LCP}[j'+1] \geq \texttt{LCP}[\ell]$. Consequently, $\texttt{w}(j'') \geq \texttt{LCP}[j'+1] \geq \texttt{LCP}[\ell] = \texttt{w}(i)$.

Finally, since $\texttt{BWT}[j''] = c$, $\texttt{curr\_b} = \texttt{b}(\ell) \leq j'' < i$, and $\texttt{w}(j'') \geq \texttt{w}(i)$, this contradicts $i = \texttt{Candt}_c(\texttt{curr\_b}, i+1)$. Hence, the assumption is false, and the lemma follows. ◀

Note that in Lemma 12, if $\texttt{w}(i) > \texttt{LCP}[i]$, then $\texttt{w}(i) = \texttt{LCP}[i+1] > \texttt{LCP}[i]$ and $\texttt{curr\_b} = \texttt{b}(i+1) = i$; hence, the interval $[\texttt{curr\_b}, i)$ is empty.

▶ **Lemma 13.** *The entries in* `rowList` *are sorted in non-increasing order by weight.*

Proof. We prove the claim by induction on the iteration index.

In the first iteration, if $\texttt{BWT}[1] = \texttt{BWT}[2]$, no triple is added to `rowList`, so it remains empty. Otherwise, `rowList` contains exactly one triple. In both cases, the claim holds trivially.

Assume as induction hypothesis that after each of the first $(i-1)$ iterations (for some $i > 1$), the triples in `rowList` are sorted in non-increasing order of their `w` values.

Now consider the i-th iteration. Any triple removed from `rowList` in Step 1 is always removed from the front and has weight strictly greater than $\texttt{LCP}[i]$. Therefore, after Step 1, the relative order of the remaining triples is preserved. Let (p_{text}, c, w) denote the triple currently at the front of `rowList` (if any); then we have $\texttt{LCP}[i] \geq w$.

If no triple is added to `rowList` in Step 2, the claim follows immediately from the induction hypothesis. Otherwise, we must show that the newly added triple $(n - \texttt{SA}[i] + 1, \texttt{BWT}[i], \texttt{w}(i))$ satisfies $\texttt{w}(i) \geq w$. Since $\texttt{LCP}[i] \geq w$, if $\texttt{w}(i) \geq \texttt{LCP}[i]$, then $\texttt{w}(i) \geq w$ holds immediately.

It remains to consider the case $\texttt{w}(i) < \texttt{LCP}[i]$. Since the triple is added at iteration i, we have $i = \texttt{Candt}_c(\texttt{curr\_b}, i+1)$ for $c = \texttt{BWT}[i]$. Because both $i = \texttt{Candt}_c(\texttt{curr\_b}, i+1)$ and $\texttt{w}(i) < \texttt{LCP}[i]$, Lemma 12 applies, so either $\texttt{BWT}[j] = c$ for every $j \in [\texttt{curr\_b}, i)$ or $\texttt{BWT}[j] \neq c$ for every such j. Since $i > 1$ and $\texttt{w}(i) < \texttt{LCP}[i]$, by definition of $\texttt{w}(\cdot)$ we must have $\texttt{BWT}[i-1] = \texttt{BWT}[i]$, which implies $\texttt{BWT}[j] = c$ for every $j \in [\texttt{curr\_b}, i)$. Consequently, $\texttt{w}(j) = -1$ for every $j \in (\texttt{curr\_b}, i)$, and either $\texttt{w}(\texttt{curr\_b}) = \texttt{LCP}[\texttt{curr\_b}]$ or $\texttt{w}(\texttt{curr\_b}) = -1$. The former case implies $\texttt{SA}^{-1}[n - p_{\text{text}} + 1] \leq \texttt{curr\_b}$. Moreover, since $\texttt{SA}^{-1}[n - p_{\text{text}} + 1] \leq \texttt{curr\_b}$ and $\texttt{w}(\texttt{curr\_b}) \leq \texttt{LCP}[\texttt{curr\_b}]$, it follows that $w \leq \texttt{LCP}[\texttt{curr\_b}]$. On the other hand, since $\texttt{w}(i) < \texttt{LCP}[i]$, we have $\texttt{w}(i) = \texttt{LCP}[i+1]$ and $\texttt{curr\_b} = \texttt{b}(i+1)$. Because $\texttt{LCP}[i+1] > \texttt{LCP}[\texttt{curr\_b}]$ and $w \leq \texttt{LCP}[\texttt{curr\_b}]$, it follows that $\texttt{w}(i) > w$.

Therefore, whether $\texttt{w}(i) \geq \texttt{LCP}[i]$ or not, we have $\texttt{w}(i) \geq w$ (in fact, strictly greater in the latter case), so inserting the new triple preserves the non-increasing order in `rowList`. This completes the induction and the proof. ◀

4.3.2 Necessity: `result` $\subseteq$ `FullL`

Throughout this section, let $\texttt{rowList}_i$ denote the state of `rowList` at the start of the i-th iteration of the algorithm. Let $\texttt{POS}_i = \{\texttt{SA}^{-1}[n - p_{\text{text}} + 1] \mid (p_{\text{text}}, \cdot, \cdot) \in \texttt{rowList}_i\}$. For each $j \in \texttt{POS}_i$, define $x_j \in \{j, j+1\}$ by $x_j = j$ if $\texttt{w}(j) = \texttt{LCP}[j]$, and $x_j = j+1$ otherwise. Observation 14 details the properties of x_j.

▶ **Observation 14.** *For each $j \in \texttt{POS}_i$, the following hold: (a) either $\texttt{BWT}[j] \neq \texttt{BWT}[j-1]$ or $\texttt{BWT}[j] \neq \texttt{BWT}[j+1]$; (b) $\texttt{w}(j) = \texttt{LCP}[x_j]$; and (c) $\texttt{b}(x_j) = \texttt{curr\_b}$ at iteration j.*

Proof. Since $j \in \texttt{POS}_i$, the algorithm guarantees $\texttt{w}(j) \geq 0$. By definition of $\texttt{w}(\cdot)$, this implies $\texttt{w}(j) \in \{\texttt{LCP}[j], \texttt{LCP}[j+1]\}$ and that either $\texttt{BWT}[j] \neq \texttt{BWT}[j-1]$ or $\texttt{BWT}[j] \neq \texttt{BWT}[j+1]$, proving (a), and $\texttt{w}(j) = \texttt{LCP}[x_j]$, proving (b).

If $\texttt{w}(j) \neq \texttt{LCP}[j]$, then $x_j = j+1$ and $\texttt{curr\_b} = \texttt{b}(j+1)$, so $\texttt{curr\_b} = \texttt{b}(x_j)$ at iteration j. Otherwise, $x_j = j$ and $\texttt{curr\_b} = \texttt{b}(j)$, so again $\texttt{curr\_b} = \texttt{b}(x_j)$. This proves (c). ◀

▶ **Lemma 15.** *For every $j \in \texttt{POS}_i$, if $\texttt{w}(j) \neq \texttt{LCP}[j]$, then $\texttt{Candt}_c(\texttt{b}(j+1), i) = j$, where $c = \texttt{BWT}[j]$; otherwise, $\texttt{Candt}_c(\texttt{b}(j), i) = j$.*

Proof. Consider the j-th iteration of the algorithm. Since $j \in \texttt{POS}_i$ and $j < i$, the tuple $(n - \texttt{SA}[j] + 1, \texttt{BWT}[j], \texttt{w}(j))$ is added to $\texttt{rowList}$ in the j-th iteration. As shown in the algorithm, we know that at iteration j, the condition $j = \texttt{Candt}_c(\texttt{curr\_b}, j+1)$ holds; moreover, by Observation 14, we have $\texttt{curr\_b} = \texttt{b}(x_j)$, so, $j = \texttt{Candt}_c(\texttt{b}(x_j), j+1)$.

Since $j \in \texttt{POS}_i$, we know that for any index $j < j' < i$, $\texttt{LCP}[j']$ cannot be smaller than $\texttt{w}(j)$ (otherwise, the tuple with *weight* $= \texttt{w}(j)$ would be removed from $\texttt{rowList}$ before iteration i); moreover, if $\texttt{BWT}[j'] = c$, then $\texttt{w}(j') \leq \texttt{w}(j)$. Therefore, we have $j = \texttt{Candt}_c(\texttt{b}(x_j), i)$. If $\texttt{w}(j) \neq \texttt{LCP}[j]$, then $x_j = j+1$, so $\texttt{Candt}_c(\texttt{b}(j+1), i) = j$; otherwise, $x_j = j$, and $\texttt{Candt}_c(\texttt{b}(j), i) = j$, completing the proof. ◀

▶ **Lemma 16.** *For every $j \in \texttt{POS}_i$ with $\texttt{LCP}[i] < \texttt{w}(j)$, if $\texttt{w}(j) \neq \texttt{LCP}[j]$, then $\texttt{e}(j+1) = i$; otherwise, $\texttt{e}(j) = i$.*

Proof. Since $j \in \texttt{POS}_i$, we know that $\texttt{LCP}[j'] \geq \texttt{w}(j)$ for any index $j < j' < i$ (otherwise, $j \notin \texttt{POS}_i$, a contradiction). Recall that $\texttt{w}(j) = \texttt{LCP}[x_j]$, where $x_j \in \{j, j+1\}$, by Observation 14. Since $\texttt{w}(j) > \texttt{LCP}[i]$ and $\texttt{w}(j) = \texttt{LCP}[x_j]$, we have $\texttt{LCP}[x_j] > \texttt{LCP}[i]$, so $i > x_j$ is the smallest index with $\texttt{LCP}[i] < \texttt{LCP}[x_j]$, so, we have $\texttt{e}(x_j) = i$. If $\texttt{w}(j) \neq \texttt{LCP}[j]$, then $\texttt{w}(j) = \texttt{LCP}[j+1]$ and $x_j = j+1$, so $\texttt{e}(j+1) = i$; otherwise, $\texttt{w}(j) = \texttt{LCP}[j]$ and $x_j = j$, so $\texttt{e}(j) = i$. ◀

Lemma 17 and Definition 7 imply $\ell_{\text{text}} \in \texttt{FullL}$ for any $\ell_{\text{text}} \in \texttt{result}_c$ and any $c \in [1, \sigma]$.

▶ **Lemma 17.** *Let $(p_{text}, c, weight)$ be any triple removed from $\texttt{rowList}_i$ in the first step of the i-th iteration of the algorithm, and let $j = \texttt{SA}^{-1}[n - p_{text} + 1]$. Either $\texttt{Candt}_c(\texttt{b}(j+1), \texttt{e}(j+1)) = j$ and $\texttt{BWT}[j+1] \neq \texttt{BWT}[j]$, or $\texttt{Candt}_c(\texttt{b}(j), \texttt{e}(j)) = j$ and $\texttt{BWT}[j] \neq \texttt{BWT}[j-1]$.*

Proof. Since p_{text} is removed from $\texttt{rowList}_i$, we have $\texttt{w}(j) \geq 0$. Thus, either $\texttt{w}(j) = \texttt{LCP}[j]$ or $\texttt{w}(j) = \texttt{LCP}[j+1] \neq \texttt{LCP}[j]$.

If $\texttt{w}(j) \neq \texttt{LCP}[j]$, then this implies $\texttt{BWT}[j+1] \neq \texttt{BWT}[j]$, by the definition of $\texttt{w}(\cdot)$. By Lemmas 15 and 16, we obtain $\texttt{Candt}_c(\texttt{b}(j+1), \texttt{e}(j+1)) = j$; the statement holds.

Otherwise, $\texttt{w}(j) = \texttt{LCP}[j]$, this implies that either $\texttt{BWT}[j] \neq \texttt{BWT}[j-1]$, or $\texttt{BWT}[j-1] = \texttt{BWT}[j] \neq \texttt{BWT}[j+1]$ and $\texttt{LCP}[j+1] = \texttt{LCP}[j]$. In both cases, Lemmas 15 and 16 imply $\texttt{Candt}_c(\texttt{b}(j), \texttt{e}(j)) = j$. Since $\texttt{Candt}_c(\texttt{b}(j), \texttt{e}(j)) = j$, if $\texttt{BWT}[j] \neq \texttt{BWT}[j-1]$, then the statement holds immediately. Otherwise, since $\texttt{LCP}[j+1] = \texttt{LCP}[j]$, the definitions of $\texttt{b}(\cdot)$ and $\texttt{e}(\cdot)$ imply $\texttt{b}(j+1) = \texttt{b}(j)$ and $\texttt{e}(j+1) = \texttt{e}(j)$, and hence $\texttt{Candt}_c(\texttt{b}(j+1), \texttt{e}(j+1)) = j$. So, we have $\texttt{BWT}[j] \neq \texttt{BWT}[j+1]$ and $\texttt{Candt}_c(\texttt{b}(j+1), \texttt{e}(j+1)) = j$, completing the proof. ◀

4.3.3 Sufficiency: $\texttt{FullL} \subseteq \texttt{result}$

Throughout this section, let x be any index with $1 < x \leq n$ such that $\texttt{BWT}[x] \neq \texttt{BWT}[x-1]$. Let $c \in \{\texttt{BWT}[x], \texttt{BWT}[x-1]\}$, and define $\ell_{x,c} := \texttt{Candt}_c(\texttt{b}(x), \texttt{e}(x))$, so $\ell_{x,c} \in [\texttt{b}(x), \texttt{e}(x))$.

By Definition 7, it suffices to prove that $n - \texttt{SA}[\ell_{x,c}] + 1 \in \texttt{result}_c$. To this end, we first prove in Lemma 18 that the triple $(n-\texttt{SA}[\ell_{x,c}]+1, c, \texttt{w}(\ell_{x,c}))$ is added to $\texttt{rowList}$ in the $\ell_{x,c}$-th iteration of the algorithm. We then show in Lemma 19 that the tuple $(n-\texttt{SA}[\ell_{x,c}]+1, c, \texttt{w}(\ell_{x,c}))$ is removed from $\texttt{rowList}$ in the $\texttt{e}(\ell_{x,c})$-th iteration and subsequently added to $\texttt{result}_c$, which completes the proof. Finally, we present in Lemma 20 that the ordering of the indices in the output list is consistent with the suffixient array.

▶ **Lemma 18.** *The tuple* $(n - \texttt{SA}[\ell_{x,c}] + 1, c, \texttt{w}(\ell_{x,c}))$ *is added to the front of* $\texttt{rowList}$ *during the* $\ell_{x,c}$*-th iteration of the algorithm.*

Proof. Define $\ell \in \{\ell_{x,c}, \ell_{x,c}+1\}$ as follows: let $\ell = \ell_{x,c}$ if $\texttt{w}(\ell_{x,c}) = \texttt{LCP}[\ell_{x,c}]$, and $\ell = \ell_{x,c}+1$ otherwise. By construction, $\texttt{w}(\ell_{x,c}) = \texttt{LCP}[\ell]$, and at iteration $\ell_{x,c}$ we have $\texttt{curr\_b} = \texttt{b}(\ell)$.

Since $\ell \in \{\ell_{x,c}, \ell_{x,c}+1\}$, it follows that $\texttt{b}(\ell) \leq \ell_{x,c}$. Moreover, by Observation 6, $\texttt{w}(\ell_{x,c}) \geq \texttt{LCP}[x]$. Together with $\texttt{w}(\ell_{x,c}) = \texttt{LCP}[\ell]$, this implies $\texttt{LCP}[\ell] \geq \texttt{LCP}[x]$. Because $\texttt{b}(x) \leq \ell_{x,c}$, $\texttt{b}(\ell) \leq \ell_{x,c}$, and $\texttt{LCP}[\ell] \geq \texttt{LCP}[x]$, we obtain $\texttt{b}(\ell) \geq \texttt{b}(x)$. Consequently, $\ell_{x,c} \in [\texttt{b}(\ell), \ell_{x,c}+1) \subseteq [\texttt{b}(x), \texttt{e}(x))$. Therefore, $\ell_{x,c} = \texttt{Candt}_c(\texttt{b}(\ell), \ell_{x,c}+1)$. Since $\texttt{curr\_b} = \texttt{b}(\ell)$ at iteration $\ell_{x,c}$, we conclude that $\ell_{x,c} = \texttt{Candt}_c(\texttt{curr\_b}, \ell_{x,c}+1)$. Hence, the tuple $(n - \texttt{SA}[\ell_{x,c}] + 1, c, \texttt{w}(\ell_{x,c}))$ is added to the front of $\texttt{rowList}$ during the $\ell_{x,c}$-th iteration. ◀

Lemma 19 establishes that $\texttt{SA}[\ell_{x,c}] + 1 \in \texttt{result}_c$, and thus $\texttt{FullL} \subseteq \texttt{result}$.

▶ **Lemma 19.** *The position* $n - \texttt{SA}[\ell_{x,c}] + 1$ *belongs to* $\texttt{result}_c$.

Proof. By Lemma 18, the tuple $(n - \texttt{SA}[\ell_{x,c}] + 1, c, \texttt{w}(\ell_{x,c}))$ is added to the front of $\texttt{rowList}$ at the $\ell_{x,c}$-th iteration. Let $j > \ell_{x,c}$ be the smallest index such that $\texttt{LCP}[j] < \texttt{w}(\ell_{x,c})$. By Observation 6, we have $\texttt{w}(\ell_{x,c}) \geq \texttt{LCP}[x]$, and by the definition of $\texttt{e}(x)$, $\texttt{LCP}[x] > \texttt{LCP}[\texttt{e}(x)]$. Hence, $\texttt{w}(\ell_{x,c}) > \texttt{LCP}[\texttt{e}(x)]$, which implies $\ell_{x,c} < j \leq \texttt{e}(x)$.

We first show that the tuple $(n - \texttt{SA}[\ell_{x,c}] + 1, c, \texttt{w}(\ell_{x,c}))$ remains in $\texttt{rowList}$ throughout all iterations j' with $\ell_{x,c} < j' < j$. Indeed, for any such iteration j', since $\texttt{LCP}[j'] \geq \texttt{w}(\ell_{x,c})$, the tuple cannot be removed in Step 1 of the algorithm. In Step 2, if $\texttt{BWT}[j'] \neq c$, the tuple is unaffected, as only the tuple pointed to by $\texttt{MAP}[\texttt{BWT}[j']]$ may be updated. If $\texttt{BWT}[j'] = c$, then $\texttt{w}(j') \leq \texttt{w}(\ell_{x,c})$, because $\ell_{x,c} = \texttt{Candt}_c(\texttt{b}(x), \texttt{e}(x))$ and $\ell_{x,c} < j' < j \leq \texttt{e}(x)$. Assuming for contradiction that $j' = \texttt{Candt}_c(\texttt{curr\_b}, j'+1)$ at iteration j', we would have $\texttt{curr\_b} > \ell_{x,c}$. Moreover, $\texttt{LCP}[\texttt{curr\_b}] < \texttt{w}(j')$, since either $\texttt{w}(j') = \texttt{LCP}[j']$ and $\texttt{curr\_b} = \texttt{b}(j')$, or $\texttt{w}(j') = \texttt{LCP}[j'+1]$ and $\texttt{curr\_b} = \texttt{b}(j'+1)$. This yields $\ell_{x,c} < \texttt{curr\_b} \leq j' < j$ and $\texttt{LCP}[\texttt{curr\_b}] < \texttt{w}(j') \leq \texttt{w}(\ell_{x,c})$, contradicting the minimality of j. Therefore, the tuple is not removed in any iteration j' with $\ell_{x,c} < j' < j$.

If $j \leq n$, then the tuple is removed from $\texttt{rowList}$ in Step 1 at iteration j, as $\texttt{LCP}[j] < \texttt{w}(\ell_{x,c})$, and $n - \texttt{SA}[\ell_{x,c}] + 1$ is appended to $\texttt{result}_c$. Otherwise, no such index j exists and the tuple remains in $\texttt{rowList}$ after n iterations. Then, the tuple is removed in the final step and its position entry is added to $\texttt{result}_c$. Hence, $n-\texttt{SA}[\ell_{x,c}]+1 \in \texttt{result}_c$, as claimed. ◀

Lemma 20 specifies the ordering of the output list.

▶ **Lemma 20.** *Let* $\texttt{result} = \{r_1, r_2, \ldots, r_k\}$. *Then, the prefixes* $T[1..r_1]$, $T[1..r_2]$, …, $T[1..r_k]$ *are sorted in co-lexicographical order.*

Proof. Recall that $\texttt{SA}[1..n]$ is the suffix array of $T^{rev}[1..n]$. Therefore, the prefixes $T[1..n - \texttt{SA}[1]], T[1..n - \texttt{SA}[2]], \ldots, T[1..n - \texttt{SA}[n]]$ are sorted in co-lexicographical order. This implies that for any character $c \in \Sigma$ and any positions $i, j \in \texttt{result}_c$ with $i < j$, we have $T[1..n - \texttt{SA}[i]] \prec_{\text{colex}} T[1..n - \texttt{SA}[j]]$, that is, $T[1..n - \texttt{SA}[i]]$ is co-lexicographically smaller than $T[1..n - \texttt{SA}[j]]$. Since $T[n - \texttt{SA}[i] + 1] = T[n - \texttt{SA}[j] + 1] = c$, it follows that $T[1..n - \texttt{SA}[i] + 1] \prec_{\text{colex}} T[1..n - \texttt{SA}[j] + 1]$.

Moreover, for any characters $c, c' \in \Sigma$ with $c < c'$, and for any positions $i \in \texttt{result}_c$ and $j \in \texttt{result}_{c'}$, we have $T[1..n - \texttt{SA}[i] + 1] \prec_{\text{colex}} T[1..n - \texttt{SA}[j] + 1]$.

Combining the arguments above establishes the lemma. ◀

Lemmas 17 and 19 together imply that the output list `result` is exactly `FullL`. By Lemma 11, `result` is a minimum-size suffixient set, and Lemma 20 shows that its indices are sorted consistently with the definition of suffixient array, completing the proof of the correctness.

4.4 The Adjusted Algorithm

We have shown that the algorithm in Section 4.2 always computes a suffixient array correctly. But, we have not specified how to check whether or not $i = \texttt{Candt}_c(\texttt{curr\_b}, i{+}1)$ at iteration i, where $c = \texttt{BWT}[i]$. In this section, we specify the procedures to verify $i = \texttt{Candt}_c(\texttt{curr\_b}, i{+}1)$.

In the data structure part, we construct, in addition, an array $\texttt{prevW}[1..\sigma]$ consisting of σ entries, where each entry is a pair of the form $(\mathit{index}, \mathit{weight})$ drawn from $\{1, \ldots, n\} \times \{-1, 0, \ldots, n\}$, initially set to $(0, -1)$. At iteration i, for $i > 1$, the following invariant maintains: For each $c \in \Sigma$, the field $\texttt{prevW}[c].\mathit{index}$ stores the largest index $j < i$ such that $\texttt{w}(j) > -1$ and $\texttt{BWT}[j] = c$, and $\texttt{prevW}[c].\mathit{weight}$ stores the corresponding value $\texttt{w}(j)$. If no such index exists, the entry $\texttt{prevW}[c]$ is set to $(0, -1)$.

It holds that $i = \texttt{Candt}_c(\texttt{curr\_b}, i + 1)$ if and only if at least one of the following conditions is satisfied: **C1:** $\texttt{w}(i) > \texttt{LCP}[i]$; **C2:** $\texttt{w}(i) > -1$ and $\texttt{prevW}[c].\mathit{index} < \texttt{curr\_b}$; or **C3:** $\texttt{prevW}[c].\mathit{weight} < \texttt{w}(i)$. Each condition can be checked in constant time in a one-pass setting. In the remainder of this section, we prove this equivalence.

▶ **Lemma 21.** *If $i = \texttt{Candt}_c(\texttt{curr\_b}, i + 1)$ at iteration i, then one of **C1**-**C3** holds.*

Proof. Since $i = \texttt{Candt}_c(\texttt{curr\_b}, i + 1) > -1$, Definition 5 implies that $\texttt{w}(i) > -1$. If $\texttt{w}(i) > \texttt{LCP}[i]$, then **C1** trivially holds. Otherwise, by Lemma 12, either $\texttt{BWT}[j] = c$ for every $j \in [\texttt{curr\_b}, i)$ or $\texttt{BWT}[j] \neq c$ for every $j \in [\texttt{curr\_b}, i)$. In either case, observe that $\texttt{w}(j') = -1$ for every $j' \in (\texttt{curr\_b}, i)$ with $\texttt{BWT}[j'] = c$, so $\texttt{prevW}[c].\mathit{index} \leq \texttt{curr\_b}$ at iteration i.

If $\texttt{prevW}[c].\mathit{index} < \texttt{curr\_b}$, then **C2** holds. Otherwise, $\texttt{prevW}[c].\mathit{index} = \texttt{curr\_b}$, and $\texttt{BWT}[\texttt{curr\_b}] = c$. Applying Lemma 12, the fact $\texttt{BWT}[\texttt{curr\_b}] = c$ implies that $\texttt{BWT}[j] = c$ for every $j \in [\texttt{curr\_b}, i)$, so $\texttt{prevW}[c].\mathit{weight} = \texttt{w}(\texttt{curr\_b}) = \texttt{LCP}[\texttt{curr\_b}]$.

Let $\ell := i$ if $\texttt{w}(i) = \texttt{LCP}[i]$; otherwise, $\ell := i + 1$, so $\texttt{curr\_b} = \texttt{b}(\ell)$ at iteration i and $\texttt{w}(i) = \texttt{LCP}[\ell]$. As $\texttt{LCP}[\ell] > \texttt{LCP}[\texttt{b}(\ell)] = \texttt{LCP}[\texttt{curr\_b}] = \texttt{prevW}[c].\mathit{weight}$, **C3** holds.

Overall, one of three condition must hold, completing the proof. ◀

▶ **Lemma 22.** *If at iteration i at least one of the conditions **C1**–**C3** holds, then $i = \texttt{Candt}_c(\texttt{curr\_b}, i + 1)$.*

Proof. We consider the three conditions separately.

Case C1: $\texttt{w}(i) > \texttt{LCP}[i]$. By the definition of $\texttt{w}(\cdot)$, this implies $\texttt{w}(i) = \texttt{LCP}[i + 1] > \texttt{LCP}[i]$, and hence $\texttt{curr\_b} = \texttt{b}(i + 1) = i$. Therefore, $\texttt{Candt}_c(\texttt{curr\_b}, i + 1) = \texttt{Candt}_c(i, i + 1) = i$.

Case C2: $\texttt{w}(i) > -1$ **and** $\texttt{prevW}[c].\mathit{index} < \texttt{curr\_b}$. By the invariant maintained by the data structure `prevW`, this implies that $\texttt{w}(j) = -1$ for every $j \in [\texttt{curr\_b}, i)$; otherwise, $\texttt{prevW}[c].\mathit{index}$ would be at least $\texttt{curr\_b}$. Hence, i is the smallest index in $[\texttt{curr\_b}, i]$ with nonnegative weight, and therefore $\texttt{Candt}_c(\texttt{curr\_b}, i + 1) = i$.

Case C3: $\texttt{prevW}[c].\textit{weight} < \texttt{w}(i)$**.** Suppose that neither **C1** nor **C2** holds, but **C3** does. Thus, $\texttt{w}(i) \leq \texttt{LCP}[i]$ and $\texttt{prevW}[c].\textit{index} \geq \texttt{curr\_b}$. Since $\texttt{w}(i) \leq \texttt{LCP}[i]$, Lemma 12 implies that either $\texttt{BWT}[j] = c$ for all $j \in [\texttt{curr\_b}, i)$ or $\texttt{BWT}[j] \neq c$ for all such j. In both cases, every index $j' \in (\texttt{curr\_b}, i)$ with $\texttt{BWT}[j'] = c$ satisfies $\texttt{w}(j') = -1$, which implies $\texttt{prevW}[c].\textit{index} = \texttt{curr\_b}$. As $\texttt{prevW}[c].\textit{weight} = \texttt{w}(\texttt{curr\_b})$ and $\texttt{w}(i) > \texttt{prevW}[c].\textit{weight}$ by **C3**, we have $\texttt{Candt}_c(\texttt{curr\_b}, i+1) = i$. ◀

4.5 The Complexity of the Adjusted Algorithm

We first analyze the working space. By Proposition 4, the monotone stack over $\texttt{LCP}[1..n]$ contains $\mathcal{O}(h)$ triples and thus uses $\mathcal{O}(h)$ words of space, where h is the height of the suffix tree built over the reversed text. The doubly-linked list `rowList`, together with the arrays $\texttt{MAP}[1..\sigma]$ and $\texttt{prevW}[1..\sigma]$, requires $\mathcal{O}(\sigma)$ space. Finally, the lists $\texttt{result}_c$ for $c \in \Sigma$ store at most $\chi \leq n$ positions in total, where χ is the size of the suffixient array. Hence, excluding the `SA`, `LCP`, and `BWT` arrays, the overall working space is $\mathcal{O}(\chi + \sigma + h) = \mathcal{O}(\chi + h)$, since $\sigma \leq \chi$.

We now analyze the running time. The algorithm performs a single left-to-right scan of the arrays `SA`, `LCP`, and `BWT`, executing n iterations in total. The operations $\texttt{w}(\cdot)$ and $\texttt{b}(\cdot)$ are invoked at most twice per iteration, yielding $\mathcal{O}(1)$ amortized time per iteration.

By Lemma 13, the triples in `rowList` are maintained in non-increasing order of their weights. Thus, in Step 1 of the i-th iteration, each triple (p_{text}, c, w) with $w > \texttt{LCP}[i]$ can be identified and removed from the head of `rowList` in constant time.

In the second step, at most one triple is removed from `rowList` and at most one new triple is inserted at its front. Using the pointer stored in $\texttt{MAP}[\texttt{BWT}[i]]$, the triple to be removed can be located in constant time. Since at most one triple is added per iteration, at most n triples are added to and removed from `rowList` over the entire execution.

After all n iterations, the remaining triples in `rowList`, if any, are enumerated in $\mathcal{O}(\sigma)$ time. Concatenating the lists $\texttt{result}_1, \ldots, \texttt{result}_\sigma$ to produce the output takes an additional $\mathcal{O}(\sigma)$ time. Overall, the total running time is $\mathcal{O}(n + \sigma) = \mathcal{O}(n)$, assuming $\sigma \leq n$.

Combining Lemmas 17, 19, and 20 with the analysis above, we obtain the following result.

▶ **Theorem 23.** *By scanning the arrays* $\texttt{BWT}[1..n]$*,* $\texttt{SA}[1..n]$*, and* $\texttt{LCP}[1..n]$ *of the reversed input text* $T[1..n]$ *over an alphabet of size* σ *in a single pass, one can construct a suffixient array of* $T[1..n]$ *in* $\mathcal{O}(n)$ *time using* $\mathcal{O}(\chi + h)$ *words of working space in the worst case, in addition to these arrays, where* χ *denotes the size of the suffixient array and* h *denotes the height of the suffix tree built over the reversed text.*

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004. doi:10.1016/S1570-8667(03)00065-0.

2 Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

3 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms Mol. Biol.*, 14(1):13:1–13:15, 2019. doi:10.1186/S13015-019-0148-5.

4 Michael Burrows and David J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. Technical Report SRC-TR-124, Digital Equipment Corporation, Palo Alto, CA, USA, May 1994.

5 Davide Cenzato, Lore Depuydt, Travis Gagie, Sung-Hwan Kim, Giovanni Manzini, Francisco Olivares, and Nicola Prezza. Suffixient Arrays: a New Efficient Suffix Array Compression Technique, 2025. doi:10.48550/arXiv.2407.18753.

6 Luc Devroye, Wojciech Szpankowski, and Bonita Rais. A Note on the Height of Suffix Trees. *SIAM J. Comput.*, 21(1):48–53, 1992. doi:10.1137/0221005.

7 Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009. doi:10.1016/J.TCS.2009.09.012.

8 Michael L. Fredman and Dan E. Willard. Surpassing the Information Theoretic Bound with Fusion Trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

9 Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Efficient Construction of a Complete Index for Pan-Genomics Read Alignment. *J. Comput. Biol.*, 27(4):500–513, 2020. doi:10.1089/CMB.2019.0309.

10 Stefan Kurtz. Reducing the space requirement of suffix trees. *Softw. Pract. Exp.*, 29(13):1149–1171, 1999. doi:10.1002/(SICI)1097-024X(199911)29:13\%3C1149::AID-SPE274\%3E3.0.CO;2-O.

11 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinform.*, 26(5):589–595, 2010. doi:10.1093/BIOINFORMATICS/BTP698.

12 Udi Manber and Eugene W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.

13 Gonzalo Navarro, Giuseppe Romana, and Cristian Urbina. Smallest Suffixient Sets as a Repetitiveness Measure. *CoRR*, abs/2506.05638, 2025. doi:10.48550/arXiv.2506.05638.

14 Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A pangenomic index for finding maximal exact matches. *J. Comput. Biol.*, 29(2):169–187, 2022. doi:10.1089/CMB.2021.0290.

15 Esko Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

16 Peter Weiner. Linear Pattern Matching Algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.

A The Pseudocode for Computing $\mathtt{w}(i)$

Let $\mathtt{pre\_bwt} := \mathtt{BWT}[x-1]$ if $x > 1$ and -1 otherwise, $\mathtt{curr\_bwt} := \mathtt{BWT}[x]$, and $\mathtt{next\_bwt} := \mathtt{BWT}[x+1]$ if $x < n$ and -1 otherwise. Similarly, let $\mathtt{curr\_lcp} := \mathtt{LCP}[x]$, and $\mathtt{next\_lcp} := \mathtt{LCP}[x+1]$ if $x < n$, and -1 otherwise. The procedure `Compute_w` determines $\mathtt{w}(x)$ as Algorithm 1. By scanning the $\mathtt{LCP}[1..n]$ and $\mathtt{BWT}[1..n]$ arrays in a single pass, we can apply Algorithm 1 to compute $\mathtt{w}(x)$ for all $x = 1, 2, \ldots$.

Algorithm 1 `Compute_w(pre_bwt, curr_bwt, next_bwt, curr_lcp, next_lcp)`.

```
01.  is_start ← (pre_bwt ≠ −1) and (curr_bwt ≠ pre_bwt);
02.  is_end ← (next_bwt ≠ −1) and (curr_bwt ≠ next_bwt);
03.  if is_start then
04.        w_start ← curr_lcp;
05.  else
06.        w_start ← −1;
03.  if is_end then
04.        w_end ← next_lcp;
05.  else
06.        w_end ← −1;
07.  return max(w_start, w_end);
```

B The Pseudocode for Computing $\mathtt{b}(i)$

Algorithm 2 `Initial-Stack`().

```
01.  S ← an empty stack
02.  push (1, −1, −1) as (index, val, b_val) onto S
03.  return S
```

Algorithm 3 `Compute_b`$(S, \mathtt{LCP}[i], i)$.

```
01.  tuple ← S.top()
02.  if tuple.index = i then
03.          return tuple.b_val;
04.  while tuple.val ≥ LCP[i] do
05.          S.pop()
06.          tuple ← S.top()
07.  end while
08.  push (i, LCP[i], tuple.index) onto S
09.  return tuple.index
```

C The Pseudocode and an Example for Computing Suffixient Arrays

Algorithm 4 `Compute Suffixient Arrays`$(\texttt{BWT}[1..n], \texttt{SA}[1..n], \texttt{LCP}[1..n])$.

```
01. BWT[n+1] ← LCP[n+1] ← −1;
02. MAP[1..σ] ← {};
03. rowList ← doubly-linked list;
04. S ← Initial-Stack();
05. for i ← 1 to σ do
06.     result_i ← an empty list;
07. if BWT[2] ≠ BWT[1] then
08.     c ← BWT[1];
09.     append (n − SA[1] + 1, BWT[1], LCP[2]) as (p_text, char, weight) to the front of rowList;
10.     MAP[c] ← head of rowList;
11. for i ← 2 to n do
12.     ℓ ← LCP[i];                                              ▷ Step 1 starts
13.     headTriple ← rowList.head;
14.     while ℓ < headTriple.weight do
15.         append headTriple.p_text to result_{headTriple.char};
16.         remove the head entry from rowList;
17.         MAP[headTriple.char] ← null;
18.     currWeight ← Compute_w(BWT[i − 1], BWT[i], BWT[i + 1], LCP[i], LCP[i + 1]);
19.     curr_b ← Compute_b(S, LCP[i], i);                        ▷ Step 1 ends
20.     if currWeight > −1 then                                  ▷ Step 2 starts
21.         if currWeight ≠ LCP[i] then
22.             curr_b ← Compute_b(S, LCP[i + 1], i + 1);
23.         c ← BWT[i];
24.         if i = Candt_c(curr_b, i + 1) then
25.             if MAP[c] ≠ null then
26.                 triple ← MAP[c];
27.                 remove triple from rowList;
28.             append (n − SA[i] + 1, BWT[i], currWeight) to the front of rowList;
29.             MAP[c] ← head of rowList;                        ▷ Step 2 ends
30. while rowList ≠ null do                                      ▷ Final step starts
31.     headTriple ← rowList.head;
32.     append headTriple.p_text to result_{headTriple.char};
33.     remove the head entry from rowList;
34. return [result_1, result_2, ..., result_σ];                  ▷ Final step ends
```

Table 1 Execution trace of the algorithm computing a suffixient array for the input T = AGCACAGCA\$. The table provides $T^{rev}[1..n]$, the LCP, BWT, and SA arrays, the variable $curr\_b$, the contents of rowList, and the character-specific result lists $Result_c$. The final algorithm output is $[10, 1, 5, 7]$ as the suffixient array.

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**n+1**
$T[i]$	A	G	C	A	C	A	G	C	A	\$	—
$T^{rev}[i]$	A	C	G	A	C	A	C	G	A	\$	—
LCP$[i]$	-1	0	1	2	4	0	1	3	0	2	—
BWT$[i]$	A	G	G	C	\$	A	A	A	C	C	—
w(i)	0	0	2	4	4	0	-1	0	0	-1	—
SA$[i]$	10	9	4	6	1	5	7	2	8	3	—
$n -$ SA$[i] + 1$	1	2	7	5	10	6	4	9	3	8	—
LCP$[i] =$ w(i)?	F	T	F	F	T	T	F	F	T	F	—
$curr\_b$	1	1	3	4	4	1	6	1	1	9	—
w$(i) > -1$?	T	T	T	T	T	T	F	T	T	F	—
$i =$ Candt$_c(curr_b, i+1)$	T	T	T	T	T	F	N/A	F	F	N/A	—
rowList	(1,A,0)	(2,G,0) (1,A,0)	(7,G,2) (1,A,0)	(5,C,4) (7,G,2) (1,A,0)	(10,\$,4) (5,C,4) (7,G,2) (1,A,0)	(1,A,0)	(1,A,0)	(1,A,0)	(1,A,0)	(1,A,0)	—
result$_\$$					10	10	10	10	10	10	10
result$_A$											1
result$_C$					5	5	5	5	5	5	5
result$_G$					7	7	7	7	7	7	7