ulm university universität ulm

**Ulm University** | 89069 Ulm | Germany

**Faculty of Engineering, Computer Science and Psychology**
Institute of Theoretical Computer Science

# Linear-time Suffix Sorting

## A new approach for suffix array construction

Master Thesis at Ulm University

**Author:**
Uwe Baier
uwe.baier@uni-ulm.de

**Reviewers:**
Prof. Dr. Enno Ohlebusch
Prof. Dr. Uwe Schöning

**Supervisor:**
Prof. Dr. Enno Ohlebusch

2015

"Linear-time Suffix Sorting– A new approach for suffix array construction"
Version from November 11, 2015

Acknowledgements:
To my supervisor, Enno Ohlebusch, as well as to my correctors, Matthias Gerber, Annika Maier and Carolin Baier.

# Abstract

This thesis presents a new approach for linear-time suffix sorting. It introduces a new sorting principle that can be used to build the first non-recursive linear-time suffix array construction algorithm named GSACA. Although GSACA cannot hold up with the performance of state of the art suffix array construction algorithms, the algorithm introduces a lot of new ideas for suffix array construction, and therefore can be seen as a starting point for future research topics.

# Contents

# Chapter 1

# Introduction

The suffix array plays an important role in string processing and data compression. It lists the lexicographically sorted suffixes of a given text in increasing order. First described by Manber and Myers in 1990 [20] as an alternative to suffix trees [10], the suffix array nowadays is used in a wide range of applications. To name a few of the most popular ones:

- The *Burrows–Wheeler Transformation* [3] of a text can easily be obtained using the suffix array. Main applications of the BWT include lossless data compression in tools like *bzip2* [30], as well as full text indexing, a powerful method to prepare a text for fast pattern localization and a lot more operations [6].
- Another popular lossless data compression method, *Lempel–Ziv 77* [32], makes use of the suffix array for fast construction [18]. It is used in data compression tools such as *gzip* [9], and further research showed how to build a compressed text index based on Lempel–Ziv 77 [17].
- Along with the suffix array, Abouelhoda et al. introduced the *Enhanced Suffix Array* [1], one more powerful text index, which is able to remove the need for the space–intensive use of suffix trees in a lot of common string processing operations.

As one can see, the suffix array is very useful in a lot of applications. Unfortunately, constructing a suffix array from a given text turns out to be a computational hard task. Although linear–time algorithms exist, some super–linear algorithms for suffix array construction work faster in practice.

According to a survey paper of Puglisi et al. [28], suffix array construction algorithms ($SACAs$) should fulfill all of the following requirements:

- Minimal asymptotic runtime: linear–time complexity
- Fast runtime in practice, tested on real world data
- Minimal space requirements, space usage for the suffix array and the text itself in an optimal way

Although the paper was published in 2007, no SACA is able to meet all of those requirements so far, so there's still a need for an 'optimal' SACA.

As research went on, there was an increasing interest in parallel suffix array construction, as well as suffix array construction using external memory. Surprisingly, both of these areas can be combined, and perform quite better than approaches dealing only with a single case [19]. One key to this result was the use of a fast sequential SACA, so conventional SACAs could improve this technique further on.

## 1.1 Overview

As we've seen already, an 'optimal' SACA would help a lot of areas in string processing and data compression. My contribution to this theme will be a new linear–time SACA, which—as first linear–time SACA—computes the suffix array without the use of recursion.

This thesis will be organized as follows: Chapter 2 takes care of basic fundamentals of string processing and suffix array construction. In Chapter 3, the basic idea of the new algorithm will be explained, before Chapter 4 shows the algorithm along with proofs for correctness and linear runtime. Chapter 5 discusses implementation details and further optimisations, followed by performance analyses in Chapter 6. Chapter 7 finally summarizes all results and gives a lookout on future research topics.

## 1.2 Related Work

As already noted, in 1990 Manber and Myers presented the suffix array along with algorithms for its construction [20]. The algorithms needed super–linear time, but were a space–saving alternative to already existing linear–time algorithms for suffix tree construction [31, 21]. Since suffix trees require a lot of space, we will ignore algorithms for their construction and focus on linear–time SACAs instead.[1]

Later on, in 2003, four new linear–time SACAs were contemporary introduced by Kim et al. [14], Kärkkäinen and Sanders [12], Ko and Aluru [16] and Hon et al. [11]. In 2005, Joong Chae Na introduced one more linear–time SACA [25]. Amongst all of these algorithms two stood out: the well–known and elegant *Distance Cover* or *Skew* Algorithm by Kärkkäinen and Sanders [12, 13], and the *KA* Algorithm by Ko and Aluru [16] which was able to achieve good results in several benchmarks [24].

The next big step in suffix sorting history was made by Nong et al. in 2009. They invented two new algorithms using the induced sorting principle [27]. One of those algorithms, called *SA-IS* [26], was able to outperform most of the super–linear algorithms that were known to 'work good in practice' [24], while guaranteeing asymptotic linear runtime and almost optimal space requirements. As a consequence, SA-IS is the currently fastest known linear–time SACA that is able to fulfill almost all of the requirements noted by Puglisi et al. [28], and stays the 'candidate to beat'.

[1] SACAs assuming a constant–size alphabet for achieving linear–time are ignored here, too.

# Chapter 2

# Preliminaries

In this chapter, we will discuss basic definitions in string processing and some fundamentals of suffix array construction. First of all, let's begin with the term 'string' and its components.

## 2.1 String Definitions

A string typically consists of characters of a certain alphabet that can be compared to each other.

**Definition 2.1.1.** An *alphabet* $\Sigma$ is a finite set of totally ordered elements (*characters*). The size of $\Sigma$ is denoted by $\sigma = |\Sigma|$.

Let's say we got an alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Normally, one would order the elements of this alphabet in a manner like $\mathtt{a} < \mathtt{b} < \mathtt{c}$, but an other order of those elements could be possible too. However, to keep it simple, most of the time we will use the lowercase basic modern Latin alphabet, and, if not mentioned otherwise, use an 'intuitive' order over this alphabet ($\mathtt{a} < \mathtt{b} < \cdots < \mathtt{z}$). Now let's have a look at the definition of the term string.

**Definition 2.1.2.** A *string* $S$ is a finite sequence of characters over an alphabet $\Sigma$. The length of a string $S$, denoted by $|S|$, is the number of characters in the sequence. The empty string with length 0 is denoted by $\varepsilon$.

To give an example, `mississippi` is a string of length 11 over the alphabet $\Sigma = \{\mathtt{i}, \mathtt{m}, \mathtt{p}, \mathtt{s}\}$. Next, the terms substring and suffix shall be defined.

**Definition 2.1.3.** Let $S$ be a string of length $n$, and let $i$ and $j$ be integers with $1 \leq i, j \leq n$. $S[i]$ denotes the $i$-th character of the string $S$. $S[i..j]$ denotes the *substring* of $S$ starting at the $i$-th and ending at the $j$-th position. We write $S[i..j+1)$ analogous to $S[i..j]$, and state $S[i..j] = \varepsilon$ for $i > j$. Furthermore, $S_i$ denotes the $i$-th *suffix* $S[i..n]$.

To give some examples for this definitions, consider the example string $S = \mathtt{mississippi}$.

- $S[1] = \mathtt{m}$
- $S[2..1] = \varepsilon$
- $S[2..5] = S[2..6) = \mathtt{issi}$
- $S[1..11] = S_1 = \mathtt{mississippi}$

- $S[5..11] = S_5 = \texttt{issippi}$

To conclude our basic definitions, we introduce a *lexicographic order*, a way to compare and order strings.

**Definition 2.1.4.** Let $\Sigma$ be an alphabet, $S$ be a string of length $n$ over alphabet $\Sigma$ and $T$ be a string of length $m$ over alphabet $\Sigma$. We write $S <_{\mathsf{lex}} T$ and say that $S$ is *lexicographically smaller* than $T$, if one of the following conditions holds:

(i) There exists an $i$ ($1 \leq i \leq \min\{n, m\}$) with $S[i] < T[i]$ and $S[1..i) = T[1..i)$.

(ii) $S$ is a *proper prefix* of $T$, i.e. $n < m$ and $S[1..n] = T[1..n]$.

Also, we define $S =_{\mathsf{lex}} T$ if $S = T$, and write $S >_{\mathsf{lex}} T$ analogous to $T <_{\mathsf{lex}} S$, $S \leq_{\mathsf{lex}} T$ if $S <_{\mathsf{lex}} T$ or $S =_{\mathsf{lex}} T$, and $S \geq_{\mathsf{lex}} T$ analogous to $T \leq_{\mathsf{lex}} S$.

By now, the tools for introducing the suffix array are complete, but let's have a look at some examples before moving on. We again consider our example string $S = \texttt{mississippi}$ (the 'deciding character' from condition (i) of Definition 2.1.4 is highlighted):

- $S_2 = \underbrace{\texttt{ississippi} <_{\mathsf{lex}} \texttt{mississippi}}_{<} = S_1$
- $S_5 = \underbrace{\texttt{issippi} <_{\mathsf{lex}} \texttt{ississippi}}_{<} = S_2$
- $S[2..5] = \texttt{issi} <_{\mathsf{lex}} \texttt{ississippi} = S_2$

## 2.2 The Suffix Array

The suffix array of a string $S$ of length $n$ is an array containing starting positions from the suffixes $S_1, \cdots, S_n$ sorted by their lexicographic order. In other words, the suffix array lists all suffixes of a string $S$ in lexicographic order.

In order to make things easier, we introduce a special *sentinel character* \$, which is appended to the end of a string.

**Definition 2.2.1.** Let $\Sigma$ be an alphabet containing a *sentinel character* \$ with $\$ < c$ for all $c \in \Sigma \setminus \{\$\}$. Let $S$ be a string of length $n$, in which the sentinel character \$ appears exactly once at the last position (we call such a string a *nullterminated string*). The *suffix array* $\mathsf{SA}$ of $S$ is a permutation of integers in range $[1 \dots n]$ satisfying $S_{\mathsf{SA}[1]} <_{\mathsf{lex}} S_{\mathsf{SA}[2]} <_{\mathsf{lex}} \cdots <_{\mathsf{lex}} S_{\mathsf{SA}[n]}$. The *inverse suffix array* $\mathsf{ISA}$ is the inverse permutation of $\mathsf{SA}$.

Figure 2.1 shows an example of a suffix array. Clearly, the sentinel character is not needed to define the suffix array. As one can guess, when removing the sentinel character, the order of suffixes stays identical.

The advantage of the sentinel character is that we only need condition (i) of Definition 2.1.4 to compare two suffixes: Since the sentinel character is occurring only once at the end of the string, it has a different position in each suffix, i.e. if $S[i+k] = \$$, $S[j+k] \neq \$$ for any $j \neq i$. This has the consequence that no suffix $S_i$ can be a proper prefix of another suffix $S_j$, so the first condition of Definition 2.1.4 is sufficient to compare two suffixes.

| $i$ | $\mathsf{SA}[i]$ | $\mathsf{ISA}[i]$ | $S_{\mathsf{SA}[i]}$ |
|---|---|---|---|
| 1 | 12 | 6 | \$ |
| 2 | 11 | 5 | i\$ |
| 3 | 8 | 12 | ippi\$ |
| 4 | 5 | 10 | issippi\$ |
| 5 | 2 | 4 | ississippi\$ |
| 6 | 1 | 11 | mississippi\$ |
| 7 | 10 | 9 | pi\$ |
| 8 | 9 | 3 | ppi\$ |
| 9 | 7 | 8 | sippi\$ |
| 10 | 4 | 7 | sissippi\$ |
| 11 | 6 | 2 | ssippi\$ |
| 12 | 3 | 1 | ssissippi\$ |

Figure 2.1: Suffix array and inverse suffix array of the string $S =$ `mississippi$`.

## 2.3 Basic Suffix Array Construction Techniques

After introducing the suffix array, a short survey about basic suffix array construction techniques should be given here. A more detailed survey can be found in the paper of Puglisi et al. [28].

First, let's clarify why suffix array construction is computationally hard. A simple approach for suffix array construction could be to store all start positions of suffixes in an array, and then using *quicksort* to sort them. Using this approach, each comparison of two suffixes requires $O(n)$ time, so the overall time expands to $O(n^2 \log n)$, quite a bad asymptotic runtime.

Better approaches use the relation between suffixes of the same string, and are performed with three main techniques:

1. *Prefix Doubling.* Initially, suffixes get sorted by their first character, having a 'sorted context' of one character. Then, by using the current lexicographic rank of the suffix immediately following the own context, a more fine grained order of suffixes can be obtained, implying context doubling / *prefix doubling.* Thus, after repeating this step $\log n$ times, all suffixes are ordered. Each step requires $O(n)$ work, so the total time is $O(n \log n)$.

2. *Recursive.* Suffixes get typed by some criteria in a first step. Then, a new sequence is built and recursively gets sorted, so it introduces the order of suffixes of a certain type. The order of those special typed suffixes then can be used to induce the order of all suffixes. Typically, the size of the newly created sequence is smaller than $\frac{2}{3}n$, and each step except recursion can be done in $O(n)$ time, thus implying a total time of $O(n)$.

3. *Induced Copying.* The idea of typing suffixes is the same as in recursive algorithms, but instead of using recursion, efficient string sorting techniques are used to obtain the order of special typed suffixes. Then, as in recursive algorithms, this order is used to introduce the order of all suffixes. Despite super–linear worst case time those algorithms perform well in practice.

Today, all state of the art linear–time SACAs make use of recursion, achieving quite nice results in practice. It would be interesting nonetheless if a non–recursive algorithm with the capability of linear–time can be designed—at least from a theoretical point of view. This issue will be addressed by the next chapter, along with a new technique for suffix sorting.

# Chapter 3

# Algorithmic Idea

After introducing the suffix array and discussing some basic construction techniques, next goal will be to present a new sorting principle along with an algorithm to construct a suffix array in linear time. To introduce the sorting principle, a definition for the next lexicographically smaller suffix is needed first.

**Definition 3.0.1.** Let $S$ be a nullterminated string of length $n$, and let $i$ be an integer in range $[1, n)$. Then, by $\widehat{i}$ we denote the position of the next lexicographically smaller suffix of $S_i$, i.e. $\widehat{i} \coloneqq \min\{\ j \in [i+1 \dots n] \mid S_j <_{\mathsf{lex}} S_i\ \}$. Also, we define $\widehat{n} \coloneqq n+1$ for the last suffix of $S$.[1]

An example of suffixes and their next lexicographically smaller suffixes can be found in Figure 3.1.

| $i$ | $\mathsf{SA}[i]$ | $\widehat{\mathsf{SA}[i]}$ | $S_{\mathsf{SA}[i]}$ | $S[\mathsf{SA}[i]..\widehat{\mathsf{SA}[i]})$ |
|---|---|---|---|---|
| 1 | 14 | 15 | $ | $ |
| 2 | 3 | 14 | aindraining$ | aindraining |
| 3 | 8 | 14 | aining$ | aining |
| 4 | 6 | 8 | draining$ | dr |
| 5 | 13 | 14 | g$ | g |
| 6 | 1 | 3 | graindraining$ | gr |
| 7 | 4 | 6 | indraining$ | in |
| 8 | 11 | 13 | ing$ | in |
| 9 | 9 | 11 | ining$ | in |
| 10 | 5 | 6 | ndraining$ | n |
| 11 | 12 | 13 | ng$ | n |
| 12 | 10 | 11 | ning$ | n |
| 13 | 2 | 3 | raindraining$ | r |
| 14 | 7 | 8 | raining$ | r |

Figure 3.1: Suffix array and next lexicographically smaller suffixes of $S = \texttt{graindraining\$}$.

In addition to the definition of next lexicographically smaller suffixes, suffix groups will be introduced next. Roughly spoken, a suffix group is a set of suffixes sharing some prefix, but the definition will explain this more in detail.

[1] One can think of this as follows: if we define an imaginary and empty last suffix $S_{n+1} \coloneqq \varepsilon$, then $S_{n+1}$ is a proper prefix of $S_n$, so $S_{n+1}$ is the next smaller suffix of $S_n$.

**Definition 3.0.2.** Let $S$ be a nullterminated string of length $n$, and let $u$ be any substring of $S$. A *suffix group* $\mathcal{G}$ with *group prefix* $u$ is defined as a set $\mathcal{G} \subseteq [1 \dots n]$ such that $u$ is a prefix of all suffixes in $\mathcal{G}$, i.e. $i \in \mathcal{G} \Rightarrow u$ is a prefix of $S_i$.

Furthermore, let $\mathcal{G}_1, \dots, \mathcal{G}_m$ be a partition of $[1 \dots n]$ into suffix groups, such that the group prefixes of $\mathcal{G}_1, \dots, \mathcal{G}_m$ differ. For any $i, j \in [1 \dots m]$, we write $\mathcal{G}_i < \mathcal{G}_j$ if the group prefix of $\mathcal{G}_i$ is lexicographically smaller than the group prefix of $\mathcal{G}_j$. Within such a partition, group$(i)$ identifies the suffix group to which the index $i$ belongs.

An example for suffix groups can be found in Figure 3.2. When comparing suffix groups as in Definition 3.0.2, we will use the terms lower ordered (or just lower) respectively higher ordered (or just higher) instead of smaller or larger: because suffix groups are sets, the terms smaller or larger usually refer to set sizes, not to lexicographic comparison between group prefixes. Now, within this definitions, it is possible to give a first idea of the algorithm, as we will see next.

## 3.1 Basic Sorting Principle

In a first phase, suffixes will be sorted as if each suffix $S_i$ consists only of the string $S[i..\widehat{i}]$. If $S[i..\widehat{i}] = S[j..\widehat{j}]$ holds for two suffixes $S_i$ and $S_j$, they are placed in the same group, otherwise in different groups, so $S[i..\widehat{i}]$ denotes the group prefix of $i$'s group. The groups will be ordered according to their group prefixes, as Definition 3.0.2 states. An example for such a suffix grouping can be found in Figure 3.2.

| group suffixes | {14} | {3} | {8} | {6} | {13} | {1} | {11, 9, 4} | {12, 10, 5} | {7, 2} |
|---|---|---|---|---|---|---|---|---|---|
| group prefix | $ | aindraining | aining | dr | g | gr | in | n | r |

Figure 3.2: Suffix grouping of $S =$ `graindraining`\$ after Phase 1 of Algorithm 1. Groups are positioned from left lowest ordered to right highest ordered.

Then, in a second phase, this information will be used to sort suffixes finally. Think about a suffix $S_i$ with $S[\,\widehat{i}\,] =$ \$. Because of the sorting in the first phase it is clear that all suffixes in lower ordered groups are lexicographically smaller. On the other hand, since \$ is the lexicographically smallest suffix, it is clear that $S_i$ must be the lexicographically smallest suffix in its current group. Now, define $sr$ to be the number of suffixes placed in lower groups than group$(i)$, $sr \coloneqq |\{\ j \in [1 \dots n] \mid \text{group}(j) < \text{group}(i)\ \}|$, then $S_i$ is the lexicographically $sr + 1$-th smallest suffix, and we can set $\mathsf{SA}[sr + 1] \leftarrow i$. Additionally, if we remove $S_i$ from its group and put it in a new group placed immediately before $i$'s old group, the group order of Definition 3.0.2 stays consistent, and the same procedure can be repeated for the next minimal element of $S_i$'s old group.

Using this idea, the second phase will proceed like the following: first, set $\mathsf{SA}[1] \leftarrow n$ because of the definition of the sentinel character. Then, iterate the suffix array from 1 to $n$ in increasing order. Within the $i$-th iteration, compute all suffixes $S_j$ with $\widehat{j} = \mathsf{SA}[i]$, and execute the procedure described above for all of them. Some exemplar iterations can be found in Figure 3.3.

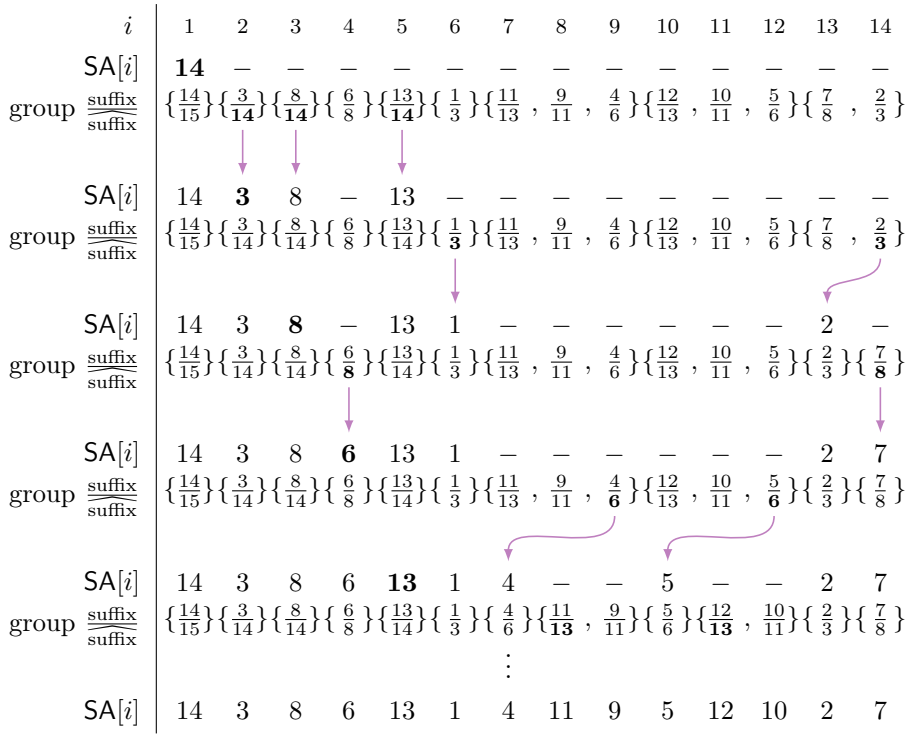



Figure 3.3: Phase 2 steps of Algorithm 1, applied to the string $S =$ `graindraining`\$. Suffixes of groups are listed along with their next lexicographically smaller suffixes, displayed as fractions.

Now finally we can give a first principle how suffixes can be sorted:

**Algorithm 1** Suffix array construction for a given nullterminated string $S$ of length $n$.

*Phase 1: sort suffixes into groups*

1: order all suffixes of $S$ into groups: Let $S_i$ and $S_j$ be two suffixes. Then, $\text{group}(i) = \text{group}(j)$ if and only if $S[i..\widehat{i}) = S[j..\widehat{j})$.

2: order the suffix groups by their group prefixes: Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two groups, $i \in \mathcal{G}_1$, $j \in \mathcal{G}_2$. Then, $\mathcal{G}_1 < \mathcal{G}_2$ if and only if $S[i..\widehat{i}) <_{\text{lex}} S[j..\widehat{j})$.

*Phase 2: construct suffix array from groups*

3: $\mathsf{SA}[1] \leftarrow n$

4: **for** $i = 1$ **up to** $n$ **do**

5: **for all** suffixes $S_j$ with $\widehat{j} = \mathsf{SA}[i]$ **do**

6: let $sr$ be the number of suffixes placed in lower groups, i.e. $sr := |\{\ s \in [1 \dots n] \mid \text{group}(s) < \text{group}(j)\ \}|$.

7: $\mathsf{SA}[sr + 1] \leftarrow j$

8: remove $j$ from its current group and put it in a new group placed as immediate predecessor of $j$'s old group.

9: **end for**

10: **end for**

Next, some more details of Algorithm 1 shall be explained. In line 2, a group $\mathcal{G}_1$ is mentioned to be lower ordered to a group $\mathcal{G}_2$ if the group prefix of $\mathcal{G}_1$ is lexicographically smaller than that of $\mathcal{G}_2$. If the group prefixes of both groups differ in at least one character, then certainly all suffixes of $\mathcal{G}_1$ are lexicographically smaller than that of $\mathcal{G}_2$. Now, as Definition 2.1.4 allows, consider the case when a group prefix is a proper prefix of another's group prefix. Within this case, it is not clear whether suffixes of $\mathcal{G}_1$ are lexicographically smaller than those of $\mathcal{G}_2$. To ensure correctness in this cases, a theorem will be expressed now.

**Theorem 3.1.1.** *Let $S$ be a nullterminated string of length $n$, and let $i$ and $j$ be two integers in range $[1 \dots n]$. If $S[i..\widehat{i})$ is a proper prefix of $S[j..\widehat{j})$, then it follows that $S_i <_{\mathsf{lex}} S_j$.*

*Proof.* Let $k \coloneqq \widehat{i} - i$. Because $S[i..\widehat{i})$ is a proper prefix of $S[j..\widehat{j})$, it is clear that $k < \widehat{j} - j$ and $S[i..i+k) = S[j..j+k)$. By Definition 3.0.1 of next lexicographically smaller suffixes, it is clear that $S_i >_{\mathsf{lex}} S_{i+k}$ and $S_j <_{\mathsf{lex}} S_{j+k}$. Now, let $u$ be a number such that $S[i+u] > S[i+k+u]$ and $S[i..i+u) = S[i+k..i+k+u)$, and let $v$ be a number such that $S[j+v] < S[j+k+v]$ and $S[j..j+v) = S[j+k..j+k+v)$. We assume that $u < v$; the cases $u = v$ and $u > v$ can be handled analogously. Since $S[i..i+u) = S[i+k..i+k+u)$, the suffix $S_i$ starts with repetitions of $S[i..i+k)$ until the $u+k$-th character, i.e.

$$S_i = S[i + (0 \mod k)]S[i + (1 \mod k)] \cdots S[i + (u + k - 1 \mod k)] \cdots$$

Also, since $S[j..j+v) = S[j+k..j+k+v)$, the suffix $S_j$ starts with repetitions of $S[j..j+k)$ until the $v+k$-th character. Because $u < v$, the suffix $S_j$ has the form

$$S_j = S[j + (0 \mod k)]S[j + (1 \mod k)] \cdots S[j + (u + k \mod k)] \cdots$$

Because $S_i$ and $S_j$ share the same prefix among the first $k$ characters, $S[i..i+k+u) = S[j..j+k+u)$ must hold. Now finally, we got $S[j+k+u] = S[j+u] = S[i+u] > S[\widehat{i}+u] = S[i+k+u]$, so $S_i <_{\mathsf{lex}} S_j$ must hold. □

Another question is if Algorithm 1 fills the suffix array entirely. More detailed, it has to be shown that before the $i$-th iteration of Phase 2 the lexicographically $i$-th suffix is placed into the suffix array. It is clear that the first suffix ($\mathsf{SA}[1]$) is correctly placed into the suffix array in line 3 because of the definition of the sentinel character. Now, let $S_j$ be the lexicographically $i$-th smallest suffix. Since $S_{\widehat{j}} <_{\mathsf{lex}} S_j$ holds, by induction, the suffix $S_{\widehat{j}}$ must already have been handled in one of the $i-1$ previous iterations. Within this iteration, $S_j$ is put in place into the suffix array in lines 5 to 9, and therefore $S_j$ is available before the $i$-th iteration.

So far, we've seen the basic sorting principle along with some thoughts for correctness, but there are still a lot of issues remaining: How can Phase 1 be implemented? What asymptotic time and space is required? How will groups be organized? But instead of answering those questions and presenting a more precise algorithm directly, the next section shows a larger running example of the final algorithm first, to get better insight onto the performed steps and to bridge the gap between the basic sorting principle and the final technical result.

## 3.2 An Introducing Example

In this section, a running example of the final algorithm is shown. Although the algorithm is not known yet, the example illustrates the relationship between the sorting principle and the final algorithm. Feel free to skip this section, the algorithm is listed in the next chapter; I, however, recommend to read the example, since it will give an idea about what's happening behind the scenes.

To stay consistent, the running example will construct the suffix array for the string $S =$ `graindraining`\$. Remember that the sorting principle in Algorithm 1 consists of two phases: In a first phase, suffixes are sorted into groups by comparing prefixes that reach until the next lexicographic smaller suffix. Then, in a second phase, this group information is used to fill the suffix array entirely. Both of these phases will performed here too, but a lot more in detail.

### Phase 1: sort suffixes into groups

First task is to sort suffixes into groups, as explained above. To achieve this goal, a technique quite similar to *prefix doubling* (see Section 2.3) will be used. We start with initial groups that are sorted by the first character of their suffixes, i.e. $S[i] = S[j] \Leftrightarrow \text{group}(i) = \text{group}(j)$ and $S[i] < S[j] \Leftrightarrow \text{group}(i) < \text{group}(j)$, see Figure 3.4.

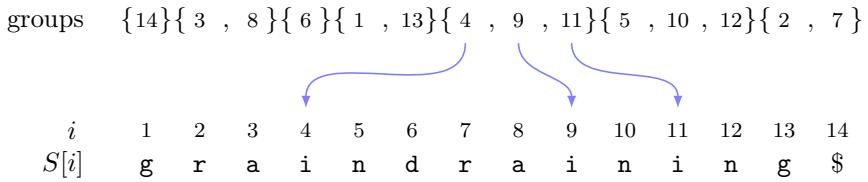
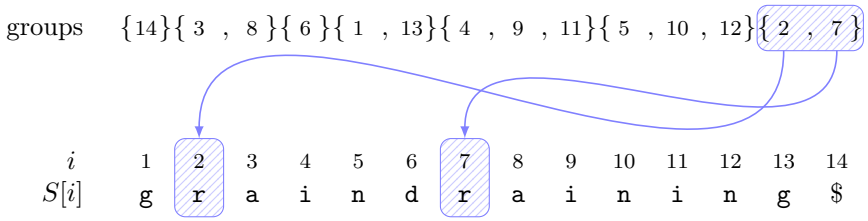


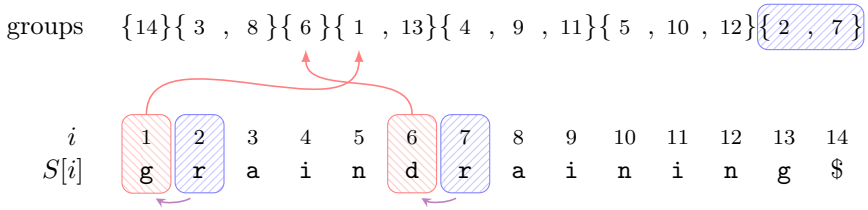

Figure 3.4: Initial groups of the string $S =$ `graindraining`\$. All suffixes sharing the same first character are placed in the same group, groups are ordered by their group prefix character from left to right. Also, links from the group with prefix `i` to its suffixes are displayed.

#### Step 1

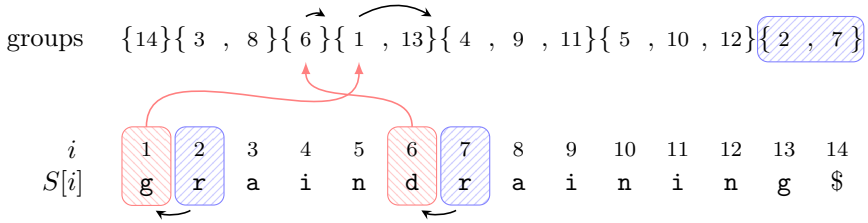
We start by processing suffixes of the highest group.



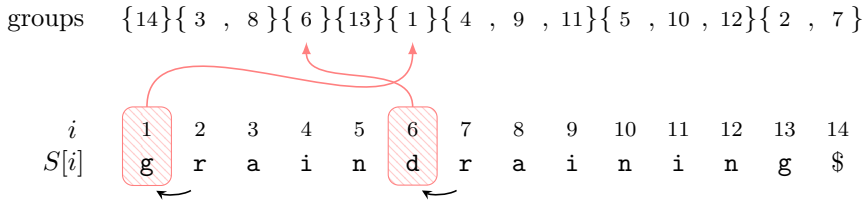
Now, for each suffix of this group, we search for the first previous suffix that is placed in a lower group. More detailed, if $S_i$ is a suffix of the current group, we search for $\text{prev}(i) := \max\{\ j \in [1..i-1] \mid \text{group}(j) < \text{group}(i)\ \}$. Additionally, pointers from suffixes to their first previous suffixes (*prev pointers*) get stored.




Next, each of those previous suffixes gets rearranged to new groups, placed immediately after their old groups. One can think of this as follows: Currently, we are working on suffixes of the highest group. Since their previous suffixes are followed by them, those previous suffixes are lexicographically larger than suffixes of the same group not followed by the processed suffixes. On the other hand, since the next higher group of such previous suffixes contains suffixes that are lexicographically larger, we have to place them in new groups immediately after their old groups.

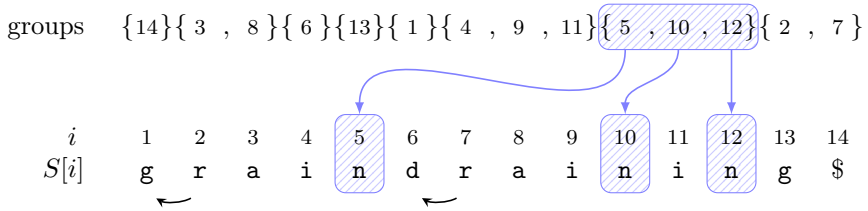



Applied to our example, we get the following rearrangements: The suffix $S_1$ is placed in a new group, between the groups $\{13\}$ and $\{4, 9, 11\}$. The suffix $S_6$ also is placed in a new group, but since the old group of $S_6$ is empty after removing $S_6$, the new group has the same position as the old one.

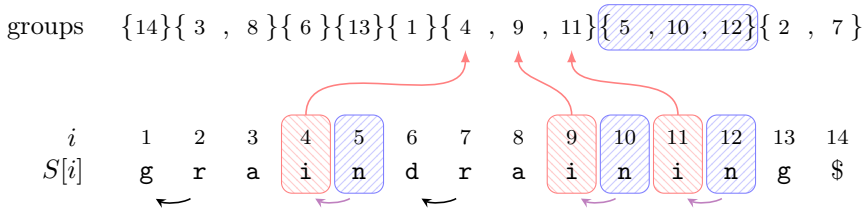



### Step 2

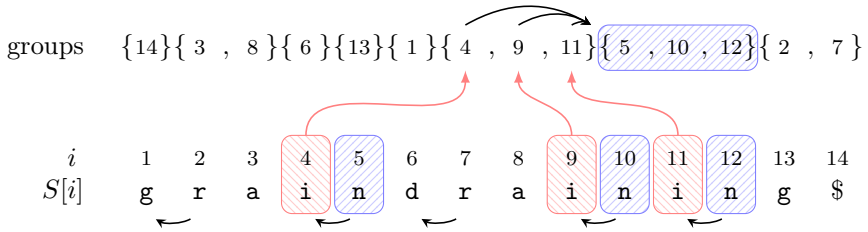
In the next step, we process the next lower group.



As in Step 1, for each suffix of the processed group we search for the first previous suffix in a lower group.

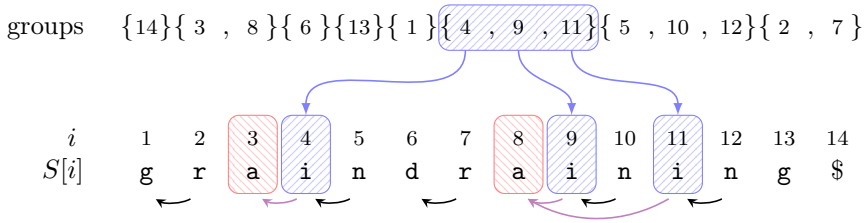



Again, those previous suffixes get rearranged in new groups, placed immediately after their old groups. In our example, all previous suffixes were placed in the same group before the rearrangement, so they'll be placed in the same group after the rearrangement. Additionally, since their old group becomes empty after the rearrangement, the new group is identical to the old one.



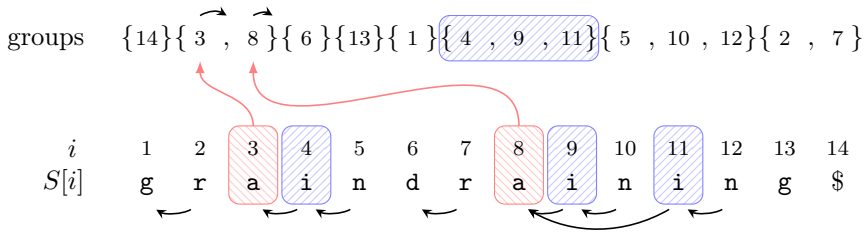

### Step 3

Again, we work on suffixes of the next lower group, and search for their first previous suffixes in lower groups. In our example, this is the first time that a previous suffix is not the immediate neighbor of a processed suffix, see $S_{11}$ and its previous suffix $S_8$.[2]



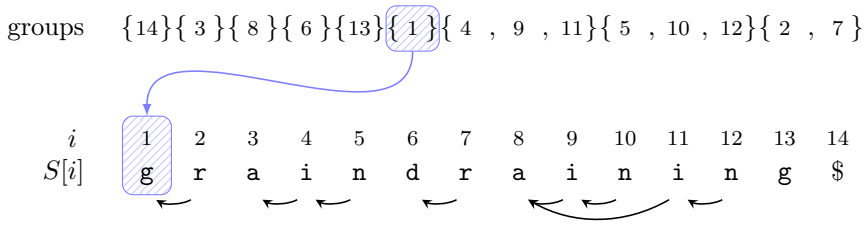

A new situation appears: The suffixes $S_3$ and $S_8$ are placed in the same group, but $S_3$ is followed by one suffix of the processed group, while $S_8$ is followed by two suffixes of the processed group. To handle this case, we can proceed as follows: If only the nearest suffixes of the processed group are used to rearrange both $S_3$ and $S_8$, then both will be placed into the same new group. After performing this step, we use the second suffix to rearrange $S_8$, so it gets placed into a new group, and therefore lands in a group higher than that of $S_3$. Consequently, $S_8$ must be rearranged to an own group that is placed higher than that of $S_3$.

[2]Note that all suffixes on the way from $S_{11}$ to $S_8$ already carry prev pointers, so they can be used to speedup the search. This technique also is known as *pointer jumping*.



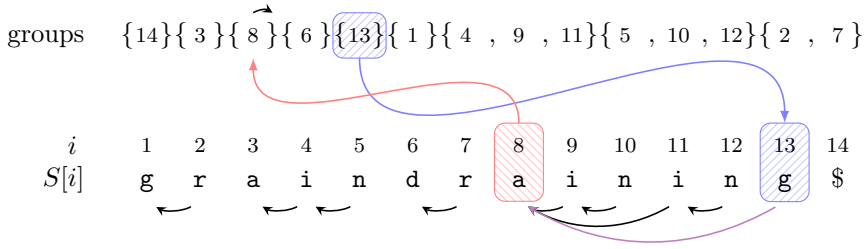

### Step 4

In the next step, the group of $S_1$ is handled. Since $S_1$ has no previous suffix in a lower group, no action is performed.




### Step 5

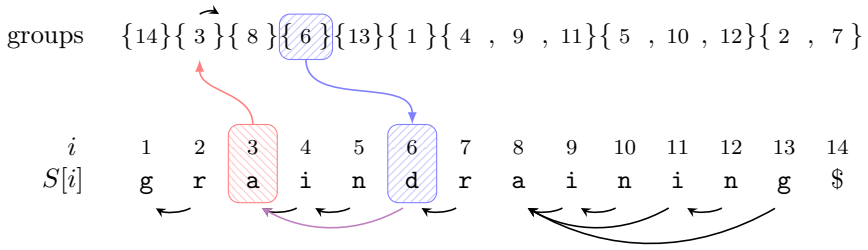
The next group to handle is that of suffix $S_{13}$. Its previous suffix is $S_8$[3], but since $S_8$ is the only suffix in its group, the rearrangement has no effect.




### Step 6

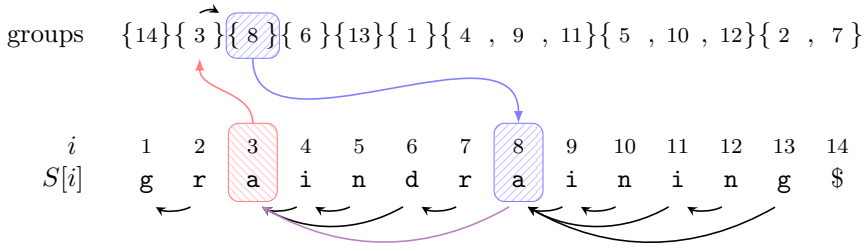
The next lower group that gets handled now is that of suffix $S_6$. As in Step 5, the previous suffix of $S_6$ has no further suffixes in its group, so the rearrangement has no effect.

[3] The pointer jumping technique mentioned in Step 3 now saves time: if we jump from $S_{12}$ to $S_{11}$, and further to $S_8$, only two operations are needed instead of four.




### Step 7

The next step has no effect, since the previous suffix of $S_8$ already is placed in a exclusive group, as in Step 6.




Although two more steps will be performed (groups {3} and {14}), they will not be shown here, because no further prev pointers get computed and so no group rearrangements take place.

### Intermediate Result

After completing Phase 1, let's have a look at the intermediate result in Figure 3.5. As one can see, the groups are identical to those of the beginning of this chapter, see Figure 3.2 on page 8. The question to be answered now is why those groups are identical.

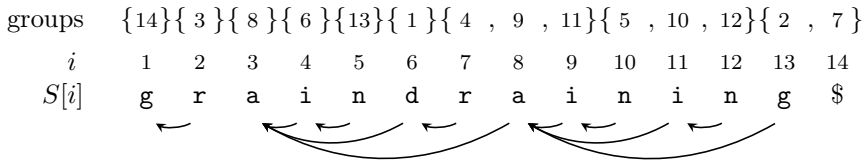



Figure 3.5: Groups and prev pointers from the example string $S =$ `graindraining$` after Phase 1.

To give a response we first need to characterize the presented algorithmic behaviour. Clearly, the algorithm behaves in a *greedy* manner, since groups were processed from highest to lowest. A hidden aspect of the algorithm is that it also works with *dynamic programming*. To explain this point, let's go back to Step 1. Before the group rearrangements, all suffixes were sorted in groups by their first character. Let's shortly denote this character as group context.

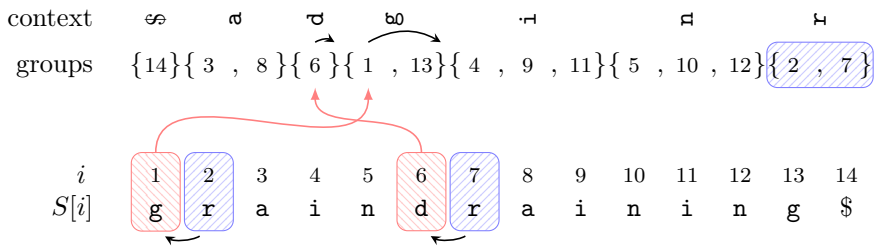



Figure 3.6: Step 1 of Phase 1, before the rearrangements take place.

Now, every time a rearrangement is performed, the group context of the processed group implicitly gets appended to that of the rearranged suffixes, thus forming new groups. Additionally, since rearrangements place suffixes between their old and their next higher groups, the order of the groups stays consistent with the lexicographic order of their contexts.

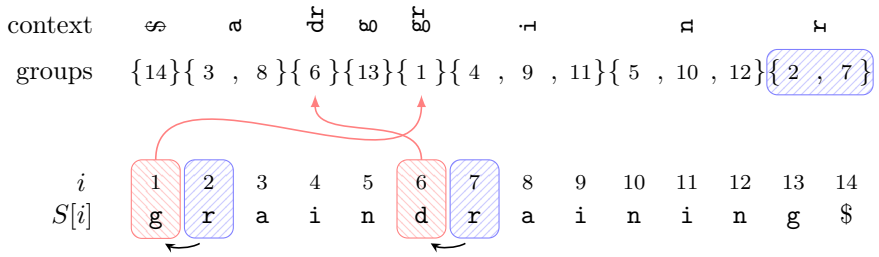



Figure 3.7: Step 1 of Phase 1, after the rearrangements took place. The context `r` of the processed group was implicitly appended to the rearranged suffixes.

This also explains the somewhat strange behaviour in Step 3: if a suffix $S_i$ of a group is followed by one group context of the currently processed group, and another suffix $S_j$ of the same group is followed by two repeated group contexts, the overall context of $S_i$ is a proper prefix of the context of $S_j$. Thus, the context of $S_i$ is lexicographically smaller than that of $S_j$, so consequently, group($i$) occurs before group($j$), see Figures 3.8 and 3.9.

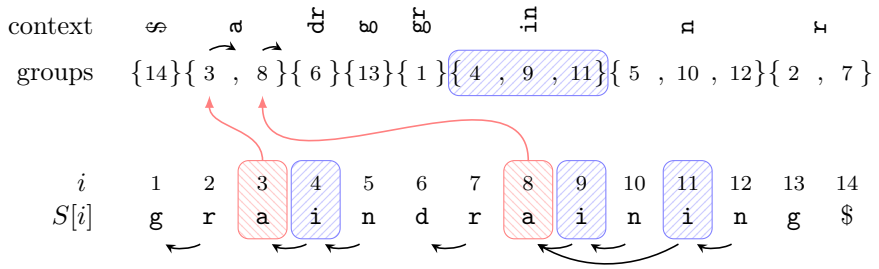



Figure 3.8: Step 3 of Phase 1, before the rearrangements.

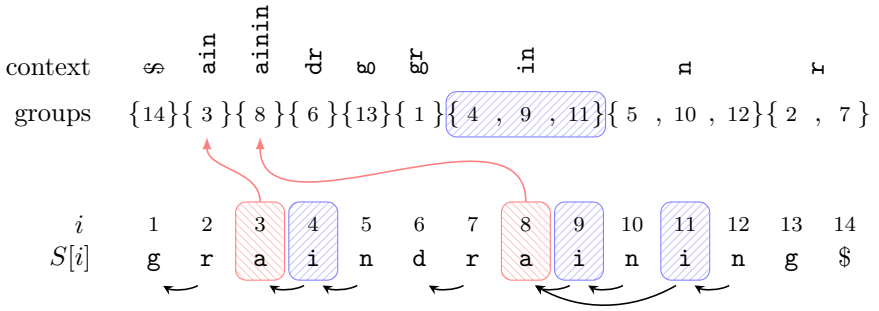



Figure 3.9: Step 3 of Phase 1, after the rearrangements. Since the new context of $S_8$ is lexicographically larger than that of $S_3$, group(3) < group(8).

This aspect of dynamic programming, or, in string context, prefix doubling, together with the greedy behaviour of the algorithm, ensures that groups are ordered as required by the sorting principle.

Now, after having handled the first phase, let's move on to the second phase to see how the suffix array can be constructed from the current information.

## Phase 2: construct suffix array from groups

After having sorted suffixes into groups, it is time to construct the suffix array. Therefore, first recall Phase 2 of the basic sorting principle, see Algorithm Excerpt 1.

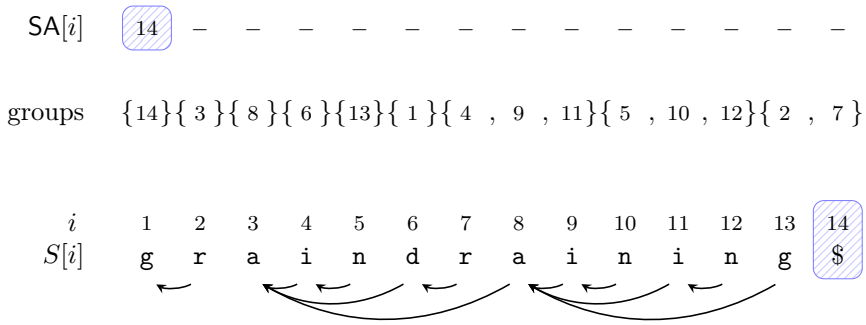
**Algorithm Excerpt 1** Phase 2 of the basic sorting principle in Algorithm 1, page 9.

```
3:  SA[1] ← n
4:  for i = 1 up to n do
5:      for all suffixes S_j with ĵ = SA[i] do
6:          let sr be the number of suffixes placed in lower groups,
            i.e. sr := |{ s ∈ [1...n] | group(s) < group(j) }|.
7:          SA[sr + 1] ← j
8:          remove j from its current group and put it in a new group
            placed as immediate predecessor of j's old group.
9:      end for
10: end for
```

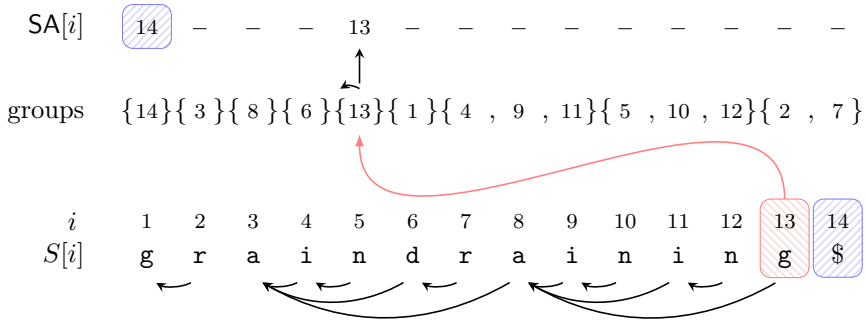
The main problem in this phase is to find, for a given suffix $S_i$, all suffixes whose next lexicographically smaller suffix equals $S_i$. More detailed, for a given suffix $S_i$, the set $\{ j \in [1 \dots n] \mid \hat{j} = i \}$ has to be computed, see line 5 of Algorithm Excerpt 1. As we shall see, the prev pointers computed in Phase 1 will handle this task for us, but let's see this step by step.

### Step 1

We start with the algorithm after line 3, because the first suffix array entry trivially is correct. So, our current suffix is $S_{14}$.

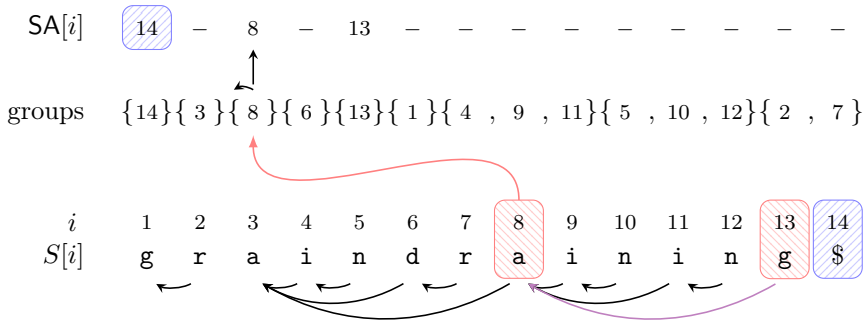



The first suffix we are going to visit is the preceding suffix of $S_{14}$, $S_{13}$. Trivially, the next lexicographically smaller suffix of $S_{13}$ is $S_{14}$. As described in the algorithm on lines 6 to 8, we remove $S_{13}$ from its group and put it in a new group placed as immediate predecessor of its old group, since it is followed by the lexicographically smallest suffix and therefore is the minimal element of its group. Also, $S_{13}$ is placed in the suffix array.

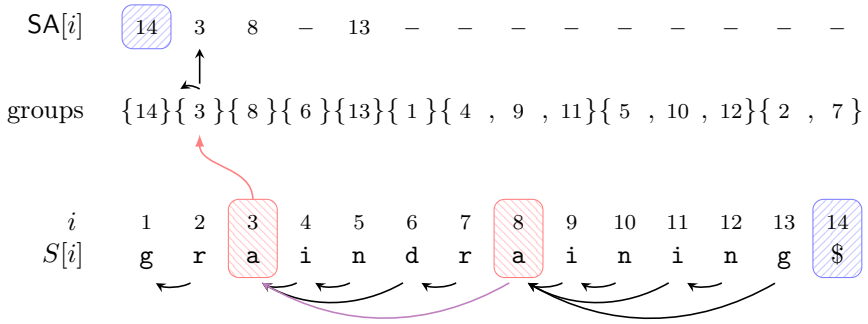



Next, we are going to follow the prev pointer from $S_{13}$ to $S_8$, repeating the same steps as above. Recall that the prev pointer by definition points to the first previous suffix placed in a lower group. Consequently, each suffix $S_j$ between $S_8$ and $S_{13}$ is placed in a group equal to or higher than that of $S_{13}$. This fact and Theorem 3.1.1 of page 10 imply that for each such suffix $S_j >_{\text{lex}} S_{13}$ holds (a proof will follow later). Thus, $\widehat{j} \leq 13$, so those suffixes must be handled in a later step.

On the other hand, since group(8) < group(13), the next lexicographically smaller suffix of $S_8$ must be behind that of $S_{13}$, i.e. $\widehat{8} \geq \widehat{13}$, so clearly, $S_{\widehat{8}} = S_{14}$. Thus, placing $S_8$ into the suffix array is correct.

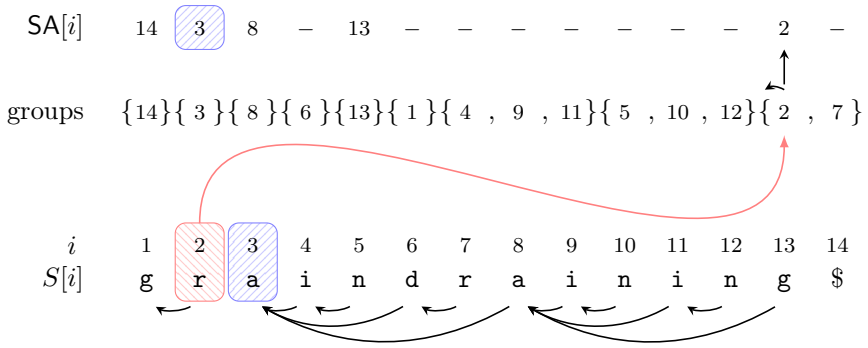


For the same reason as mentioned above, we again follow the prev pointer from $S_8$ to $S_3$, executing lines 6 to 8 of the algorithm. Since $S_3$ has no further prev pointer, the step now is complete.

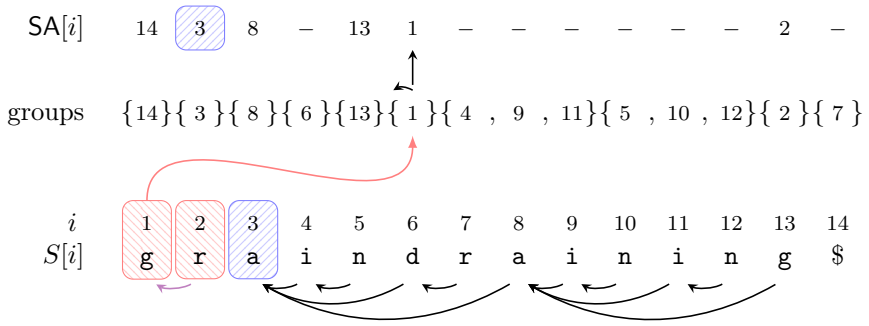



### Step 2

In the next step, we process the lexicographically 2-th smallest suffix $S_3$. Its preceding suffix is $S_2$, so we repeat the behaviour from above for $S_2$. To clarify the correctness of this behaviour: Assume $S_{\widehat{2}} \neq S_3$, so $\widehat{2} > 3$. By using the definition of next lexicographically smaller suffixes, $S_3 >_{\text{lex}} S_2$ must hold. As Phase 2 processes suffixes in increasing order, $S_2$ should have been placed in the suffix array already, but as one can verify, $S_2$ is not placed in the suffix array after step 1, so $S_{\widehat{2}} = S_3$.

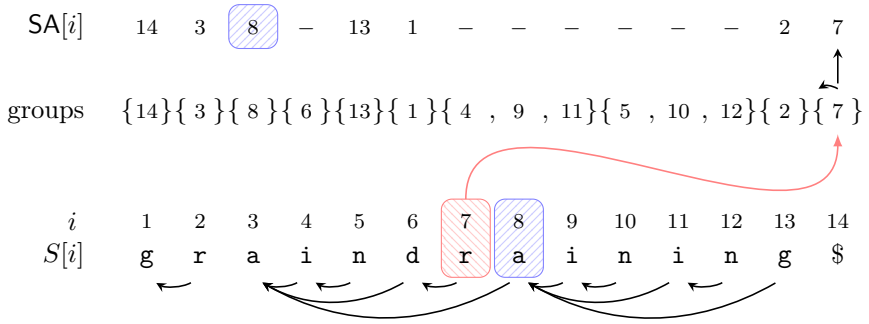



Next, we follow the prev pointer from $S_2$ to $S_1$, and place it into the suffix array. As $S_1$ has no further prev pointer to follow, the step is complete.

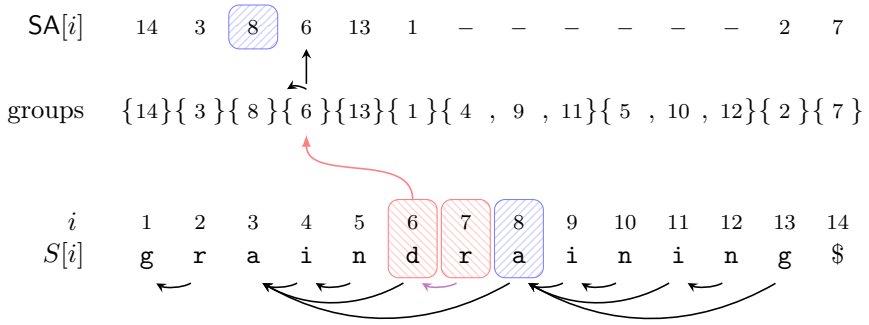



### Step 3

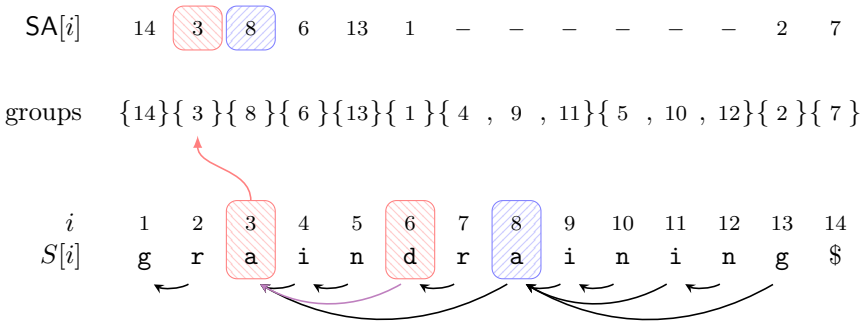
In this step, the suffix $S_8$ is processed. As before, we execute lines 6 to 8 of Algorithm Excerpt 1 with its preceding suffix $S_7$.




Next, we follow the prev pointer from $S_7$ to $S_6$, repeating the procedure.



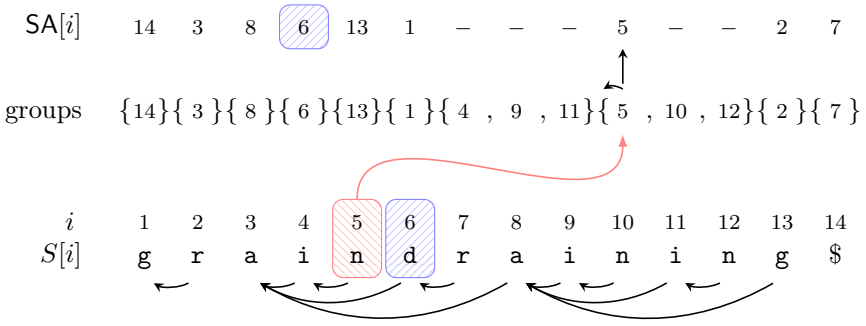

Now, when following the next prev pointer, we reach suffix $S_3$, which is already contained in the suffix array. Thus, no action needs to be performed for $S_3$. Additionally, since $S_3$ already is placed in the suffix array, any possible prev pointer from $S_3$ to a previous suffix was handled already, so we do not need to proceed.



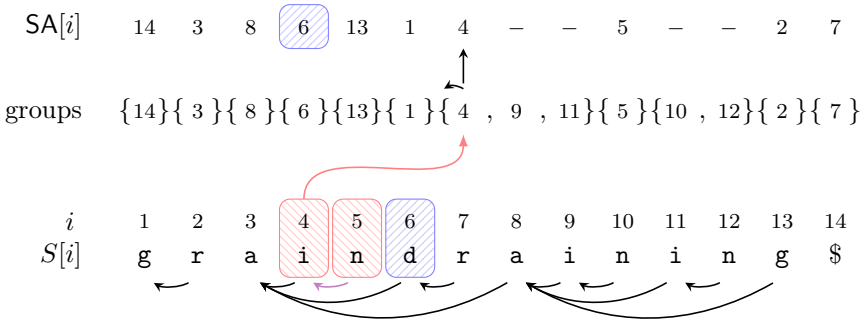

Another reason why no further suffix must be processed can be thought of as follows: suppose there exists another suffix which needs to be processed, i.e. there exists an $i < 3$ with $\widehat{i} = 8$. Then, by definition of next lexicographically smaller suffixes, $S_3 >_{\text{lex}} S_i >_{\text{lex}} S_{\widehat{i}}$ must hold. Since $\widehat{i} = 8$, $\widehat{3} \leq 8$. If $\widehat{3} < 8$, using the definition of the next lexicographically smaller suffix applied to $S[i..\widehat{i})$, $S_{\widehat{3}} >_{\text{lex}} S_8$ must hold. If $\widehat{3} = 8$, $S_{\widehat{3}} =_{\text{lex}} S_8$, so by combining both cases, $S_{\widehat{3}} \geq_{\text{lex}} S_8$ must hold. This means that the suffix $S_3$ can not have been placed into the suffix array before the current step. This leads us to a contradiction, and clearly shows that no such $i$ can exist.

### Step 4

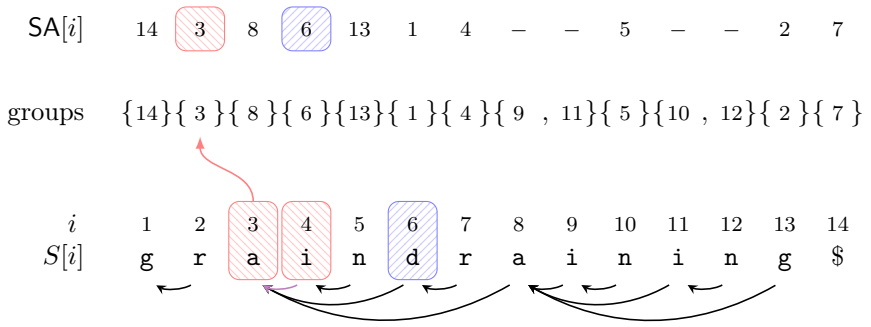
As next step, the suffix $S_6$ gets processed. As before, we start by handling its preceding suffix $S_5$.




We follow the prev pointer to $S_4$ and repeat the procedure.



Now, the next prev pointer links from $S_4$ to $S_3$, but $S_3$ is already placed in the suffix array, so we can stop here.




By now, I would like to come to an end with the running example. The steps 5 to 14 continue quite similarly: start at the processed suffix, and jump to its preceding suffix. Then, if the suffix is not contained in the suffix array, execute lines 6 to 8 of Algorithm Excerpt 1 with the suffix, follow the prev pointer if one exists and repeat the procedure. The reader might perform the steps himself/herself as a little exercise.

What we've seen so far is a sorting principle along with an running example illustrating an efficient implementation of the principle. However, an illustration cannot replace a concrete algorithm. The next chapter will finally show such an algorithm, along with a correctness proof and an analysis of the asymptotic runtime. Note that the main points were explained within this section already, so the following algorithm will not cover too much new details.

# Chapter 4

# The Algorithm

Without any more preliminaries, Algorithm 2 shows how the suffix array can be constructed using the new sorting principle (due to space limitations, the algorithm is split on two pages).

As described in Chapter 3, the algorithm consists of two phases. Note that the groups used in the algorithm are not the same as in the basic sorting principle (Algorithm 1 on page 9). Within this algorithm, the groups are build incrementally, in the same manner as described in the introducing example from the previous chapter. Nonetheless, the group definition from page 8 can be applied, so we're still able to compare groups by their representative group prefixes. The only difference is that group prefixes consist of an implicit context, as mentioned in the intermediate result after Phase 1 of the introducing example, Section 3.2. A formal definition for implicit contexts will be presented later, first let's have a look at the algorithm.

**Algorithm 2** Suffix array construction of a given nullterminated string $S$ of length $n$.

*Phase 1: sort suffixes into groups*

order all suffixes of $S$ into groups according to their first character:
Let $S_i$ and $S_j$ be two suffixes. Then, $\text{group}(i) = \text{group}(j) \Leftrightarrow S[i] = S[j]$.
order the suffix groups: Let $\mathcal{G}_1$ be a suffix group with group prefix character $u$, $\mathcal{G}_2$ be a suffix group with group prefix character $v$. Then, $\mathcal{G}_1 < \mathcal{G}_2$ if $u < v$.
**for each** group $\mathcal{G}$ in descending group order **do**
    **for each** $i \in \mathcal{G}$ **do**
        $\text{prev}(i) \leftarrow \max(\{\ j \in [1 \dots i] \mid \text{group}(j) < \text{group}(i)\ \} \cup \{0\})$
    **end for**
    let $\mathcal{P}$ be the set of previous suffixes from $\mathcal{G}$,
    $\mathcal{P} \coloneqq \{\ j \in [1 \dots n] \mid \text{prev}(i) = j \text{ for any } i \in \mathcal{G}\ \}$.
    split $\mathcal{P}$ into $k$ subsets $\mathcal{P}_1, \dots, \mathcal{P}_k$ such that a subset $\mathcal{P}_l$ contains
    suffixes whose number of prev pointers from $\mathcal{G}$ pointing to them
    is equal to $l$, i.e. $i \in \mathcal{P}_l \Leftrightarrow |\{\ j \in \mathcal{G} \mid \text{prev}(j) = i\ \}| = l$.
    **for** $l = k$ **down to** 1 **do**
        split $\mathcal{P}_l$ into $m$ subsets $\mathcal{P}_{l_1}, \dots, \mathcal{P}_{l_m}$ such that suffixes
        of same group are gathered in the same subset.
        **for** $q = 1$ **up to** $m$ **do**
            remove suffixes of $\mathcal{P}_{l_q}$ from their group and put them into a new
            group placed as immediate successor of their old group.
        **end for**
    **end for**
**end for**

```
    Phase 2: construct suffix array from groups
16: SA[1] ← n
17: for i = 1 up to n do
18:     j ← SA[i] − 1
19:     while j ≠ 0 do
20:         let sr be the number of suffixes placed in lower groups,
             i.e. sr := |{ s ∈ [1 . . . n] | group(s) < group(j) }|.
21:         if SA[sr + 1] ≠ nil then
22:             break
23:         end if
24:         SA[sr + 1] ← j
25:         remove j from its current group and put it in a new group
             placed as immediate predecessor of j's old group.
26:         j ← prev(j)
27:     end while
28: end for
```

## 4.1 Correctness

We first want to be sure that Algorithm 2 works correctly. A first step towards this goal will be to show that the suffix groups computed by Phase 1 partition suffixes in the same manner as in the first phase of the basic sorting algorithm, page 9.

The strategy will be like the following: first, we define prev pointer chains. This definition then can be used to define the implicit context of any suffix, as discussed at the beginning of this chapter. Afterwards, a lemma about context extensions will be expressed, showing that context extensions of the algorithm form an 'avalanche effect' that sorts suffixes as desired. Finally, this lemma can be used to proof the correct group structure after Phase 1.

**Definition 4.1.1.** Let $S$ be a nullterminated string of length $n$. For any $i \in [1 \dots n]$, during a certain point in time in the execution of Algorithm 2 applied to $S$, $\pi^k(i)$ denotes the $k$-th element of the *prev pointer chain* starting at index $i$:

$$\pi^k(i) \coloneqq \begin{cases} i & \text{, if } k = 0 \\ \text{prev}(\pi^{k-1}(i)) & \text{, if } k > 0,\ \pi^{k-1}(i) \neq 0 \text{ and } \text{prev}(\pi^{k-1}(i)) \neq \texttt{nil} \\ 0 & \text{, otherwise} \end{cases}$$

Furthermore, $\Pi(i)$ denotes the *prev pointer chain set* of $i$, $\Pi(i) \coloneqq \{\ \pi^k(i) \mid k \in \mathbb{N}_0\ \}$.

**Definition 4.1.2.** Let $S$ be a nullterminated string of length $n$. For any $i \in [1 \dots n)$, during a certain point in time in the execution of Algorithm 2 applied to $S$, $i_\mathsf{c}$ denotes the end of the *implicit context* of $i$, $i_\mathsf{c} \coloneqq \min\{\ j \in [i+1 \dots n] \mid i \in \Pi(j-1)\ \}$. Also, we define $n_\mathsf{c} \coloneqq n + 1$, and call $S[i..i_\mathsf{c})$ the *implicit context* of a suffix $S_i$.

Simpler spoken, the implicit context of a suffix $S_i$ is a prefix of $S_i$ that extends to the rightmost position that contains $i$ in its prev pointer chain. Note that this definition

exactly matches the more intuitive imagination of contexts from the introducing example on page 15—except that no formal definition was presented at that point.

These definitions, however, are enough to express an invariant describing the avalanche effect when processing groups in Phase 1 of Algorithm 2.

**Lemma 4.1.1.** *Let $S$ be a nullterminated string of length $n$. When applying Algorithm 2 to $S$, the following propositions hold before every processing of a group $\mathcal{G}$ in Phase 1, line 3:*

*(i)* $\text{group}(i) = \text{group}(j) \Leftrightarrow S[i..i_\mathsf{c}) =_\mathsf{lex} S[j..j_\mathsf{c}) \quad \forall i, j \in [1 \dots n]$

*(ii)* $\text{group}(i) < \text{group}(j) \Leftrightarrow S[i..i_\mathsf{c}) <_\mathsf{lex} S[j..j_\mathsf{c}) \quad \forall i, j \in [1 \dots n]$

*(iii)* *For all unprocessed suffixes* $S_i$ $(\ i \in [1 \dots n)$ *and* $\text{group}(i) \leq \mathcal{G}\ )$
$\text{group}(i_\mathsf{c}) \leq \mathcal{G}$ *and* $\mathcal{G} < \text{group}(j)\ \forall i < j < i_\mathsf{c}$

*(iv)* *For all processed suffixes* $S_i$ $(\ i \in [1 \dots n)$ *and* $\text{group}(i) > \mathcal{G}\ )$
$\text{group}(i_\mathsf{c}) \leq \text{group}(i)$ *and* $\text{group}(i) < \text{group}(j)\ \forall i < j < i_\mathsf{c}$

*Proof.* The proof is done by induction on the processed groups (lines 3 to 15).

**Induction Base:** Within lines 1 and 2 of the algorithm, suffixes are sorted and grouped by their first character. Since no prev pointers exist before the first iteration, the contexts of all suffixes consist only of their first characters, so propositions (i) and (ii) are satisfied. Because the first group $\mathcal{G}$ to be processed is the highest group, and every context has length 1, $\text{group}(i_\mathsf{c}) \leq \mathcal{G}$ holds for all $i \in [1, n)$, so (iii) and (iv) are also fulfilled.

**Induction Step:** Let $\mathcal{G}$ be the currently processed group, and let $\widetilde{\mathcal{G}}$ be the group to be processed next. We will show that propositions (i) to (iv) are satisfied when the processing of $\mathcal{G}$ has been finished.

Consider the point in time when the group $\mathcal{G}$ is processed. First, the algorithm computes prev pointers and the set $\mathcal{P}$ within lines 4 to 7. The first observation to be pointed out is the following:

$$p \in \mathcal{P} \Leftrightarrow p_\mathsf{c} \in \mathcal{G} \text{ and } \text{group}(p) < \mathcal{G} \tag{4.1}$$

If $\text{group}(p) \geq \mathcal{G}$, the prev pointers computed in line 5 would ignore $p$, thus $p \notin \mathcal{P}$. Next, assume that $\text{group}(p) < \mathcal{G}$ and $p_\mathsf{c} \notin \mathcal{G}$. Since $\text{group}(p) < \mathcal{G}$, $p$ must be an unprocessed suffix. In this case, as proposition (iii) states, $\text{group}(j) > \mathcal{G}$ for all $p < j < p_\mathsf{c}$, so any index $i \in \mathcal{G}$ cannot be placed between $p$ and $p_\mathsf{c}$. Additionally, combining proposition (iii) and precondition $p_\mathsf{c} \notin \mathcal{G}$, $\text{group}(p_\mathsf{c}) < \mathcal{G}$ must hold, so any index $i \in \mathcal{G}$ must be placed before $p$ or after $p_\mathsf{c}$. In the first case ($i < p$), the prev pointer computation of $i$ will ignore $p$, thus $\text{prev}(i) \neq p$. In the second case ($p_\mathsf{c} < i$), the prev pointer computation of $i$ stops at index $p_\mathsf{c}$ the latest because of $\text{group}(p_\mathsf{c}) < \mathcal{G}$, so $\text{prev}(i) \neq p$. So clearly, $p \notin \mathcal{P}$ if $\text{group}(p) < \mathcal{G}$ and $p_\mathsf{c} \notin \mathcal{G}$.

For the backward direction, let $p$ be an index with $\text{group}(p) < \mathcal{G}$ and $p_\mathsf{c} \in \mathcal{G}$. By using proposition (iii) for $p$ before the processing of group $\mathcal{G}$, it is clear that $\text{group}(j) > \mathcal{G}$ for all $j$ with $p < j < p_\mathsf{c}$. Thus, line 5 computes a prev pointer to $p$, so clearly, $p \in \mathcal{P}$.

Now, consider the point in time after the prev pointer computation. Since the algorithm has computed new prev pointers, the contexts of all suffixes in $\mathcal{P}$ are extended, see Definition 4.1.2. For any $p \in \mathcal{P}$, let $i$ be the rightmost index such that $\text{prev}(i) = p$ and $i \in \mathcal{G}$. Because $i$ is the rightmost index, using proposition (iii), $i_\mathsf{c}$ cannot be placed in group $\mathcal{G}$, otherwise a prev pointer from $i_\mathsf{c}$ to $p$ would exist, so $i$ wouldn't have been the rightmost index. As a consequence, $\text{group}(i_\mathsf{c}) < \mathcal{G}$ holds. Also, by definition of prev pointers, $\text{group}(p) < \mathcal{G}$ must hold. By the manner in which rearrangements are performed (lines 9 to 14), this statement holds even after the rearrangements. Summing everything up, after the processing of group $\mathcal{G}$, for every $p \in \mathcal{P}$, $\text{group}(p) < \mathcal{G}$, $\text{group}(p_\mathsf{c}) < \mathcal{G}$, and, because of proposition (iii) applied to all appended contexts of $p$, $\text{group}(j) \geq \mathcal{G}$ for all $p < j < p_\mathsf{c}$. Recall that Phase 1 processes groups in descending group order. Since $\widetilde{\mathcal{G}}$ is the immediate predecessor of $\mathcal{G}$, $\widetilde{\mathcal{G}} < \mathcal{G}$, and thus, proposition (iii) holds for all $p \in \mathcal{P}$ after the group $\mathcal{G}$ has been processed.

So far, we've seen that all unprocessed suffixes $S_i$ with $i_\mathsf{c} \in \mathcal{G}$ from (4.1) fulfill proposition (iii). Next, we'll have a look at unprocessed suffixes $S_i$ with $\text{group}(i) < \mathcal{G}$ and $i_\mathsf{c} \notin \mathcal{G}$. The contexts of those suffixes are not extended during the iteration. Thus, after the processing of group $\mathcal{G}$, $i_\mathsf{c}$ is not changed. Since $i_\mathsf{c} \notin \mathcal{G}$, $\text{group}(i_\mathsf{c}) < \mathcal{G}$ must hold. Because $\widetilde{\mathcal{G}}$ is the immediate predecessor of $\mathcal{G}$, $\text{group}(i_\mathsf{c}) \leq \widetilde{\mathcal{G}}$ and $\text{group}(i) \leq \widetilde{\mathcal{G}}$ holds. Also, by using proposition (iii) at the time before $\mathcal{G}$ was processed, $\text{group}(j) > \widetilde{\mathcal{G}}$ holds for all $i < j < i_\mathsf{c}$. Thus, proposition (iii) is satisfied for all of those suffixes, too.

Next, let's care about already processed suffixes. Because the next processed group $\widetilde{\mathcal{G}}$ is a predecessor of $\mathcal{G}$, and the algorithm does not change contexts of already processed suffixes, proposition (iv) is still correct after processing group $\mathcal{G}$. The remaining part is to show that the current group $\mathcal{G}$ fulfills proposition (iv) after it has been processed. Therefore, let $i \in \mathcal{G}$ be any currently processed suffix. Since $\text{group}(i) = \mathcal{G} > \widetilde{\mathcal{G}}$, proposition (iv) has to be shown. When $i$ is processed, by proposition (iii), $\text{group}(i_\mathsf{c}) \leq \mathcal{G} = \text{group}(i)$ holds. Also, for all $j$ with $i < j < i_\mathsf{c}$, $\text{group}(j) > \mathcal{G} = \text{group}(i)$ is satisfied, so proposition (iv) is fulfilled for $i$.

Summing up the current results, propositions (iii) and (iv) are correct after processing of the group $\mathcal{G}$. The next task is to show that suffixes with the same context belong to the same group, as well as to show that groups are ordered by their group prefixes[1], as propositions (i) and (ii) state.

More detailed, it has to be shown that the suffixes in a group created in line 12 share the same context, as well as to ensure the correct placement of the new groups. By our previous arguments we already know that the algorithm performs context extensions only for suffixes contained in the set $\mathcal{P}$. As a consequence, the contexts of all other suffixes remain unchanged, so it's sufficient to show that the new groups are correctly set up.

Therefore, let $u$ be the group prefix of group $\mathcal{G}$, $p \in \mathcal{P}$ be an index and $p_\mathsf{c}$ be the end of the context of $p$ before the extension. After the prev pointer computation in lines 4 to 7, the algorithm splits $\mathcal{P}$ into subsets $\mathcal{P}_1, \ldots, \mathcal{P}_k$, line 8. After the split, the following holds:

$$p \in \mathcal{P}_l \Leftrightarrow p \text{ has new context } S[p..p_\mathsf{c})u^l \tag{4.2}$$

The reason for this is the following: since $l$ equals the number of prev pointers pointing

[1] See Definition 3.0.2 on page 8. In this context, group prefixes are assumed to be the implicit contexts of the suffixes in the group.

from $\mathcal{G}$ to $p$, the context of $p$ is extended by exactly $l$ contexts of $\mathcal{G}$, thus the new context of $p$ is $S[p..p_\mathsf{c})u^l$.

Next, the algorithm processes the sets $\mathcal{P}_1, \ldots, \mathcal{P}_k$ in descending order, splits suffixes of each subset into smaller subsets such that suffixes belonging to the same group are gathered together in the same subset, and creates new groups, lines 9 to 14. This has the consequence that new groups consist of suffixes with the same extended context, so proposition (i) holds after the iteration.

The last remaining part is to show that the order of the new groups is correct, proposition (ii). Let $p \in \mathcal{P}_l$ be an index, $p_\mathsf{c}$ be the end of the context of $p$ before the context extensions, and $u$ be the group prefix of $\mathcal{G}$. After the context extensions, $S[p..p_\mathsf{c}) <_\mathsf{lex} S[p..p_\mathsf{c})u^l$, so the new group for $p$ must be placed higher than its old group, as line 12 of the algorithm performs.

Now, let $i$ be the index of a suffix of the immediate successor of $p$'s old group. If $i$ was placed in the same group as $p$ before the iteration, $i$'s context must have been extended within this iteration. Then, since the subsets $\mathcal{P}_1, \ldots, \mathcal{P}_k$ are processed in decreasing order, we know that $S[i..i_\mathsf{c}) = S[p..p_\mathsf{c})u^{\tilde{l}}$ for some $\tilde{l} > l$, so the context of $p$ is lexicographically smaller than that of $i$, and the group placement is correct. If $i$ was not placed in the same group as $p$ before the iteration, we know that the group of $i$ is higher ordered than that of $p$. Using proposition (ii) before the current iteration, $S[p..p_\mathsf{c}) <_\mathsf{lex} S[i..i_\mathsf{c})$ holds. Then, if $S[p..p_\mathsf{c})$ is no proper prefix of $S[i..i_\mathsf{c})$, $S[p..p_\mathsf{c})u^l <_\mathsf{lex} S[i..i_\mathsf{c})$ holds, thus the new group of $p$ must be placed lower than the group of $i$.

If $S[p..p_\mathsf{c})$ is a proper prefix of $S[i..i_\mathsf{c})$, consider the case when $S[p..p_\mathsf{c})u^l$ is no proper prefix of $S[i..i_\mathsf{c})$; otherwise $S[p..p_\mathsf{c})u^l <_\mathsf{lex} S[i..i_\mathsf{c})$ holds already. Now, let $\tilde{l}$ be a number such that $S[p..p_\mathsf{c})u^{\tilde{l}}$ is a proper prefix of $S[i..i_\mathsf{c})$, but $S[p..p_\mathsf{c})u^{\tilde{l}+1}$ is no proper prefix of $S[i..i_\mathsf{c})$. Also, define $j \coloneqq i + p_\mathsf{c} - p + \tilde{l}|u|$. Using propositions (iii) and (iv), we know that $\mathcal{G} < \text{group}(j)$ holds. Thus, $u <_\mathsf{lex} S[j..j_\mathsf{c})$ must hold because of proposition (ii). Also, since $u \neq S[j..j+|u|)$ holds by precondition, $u$ cannot be a proper prefix of $S[j..j_\mathsf{c})$. As we've seen in statement (4.2), contexts are extended by full contexts of other groups, so $j_\mathsf{c} < i_\mathsf{c}$ must hold. Thus, there must exist a $k$ with $k < i_\mathsf{c} - i$ and $k < p_\mathsf{c} - p + l|u|$ such that $S[p..p+k) = S[i..i+k)$ and $S[p..p+k] < S[i..i+k]$. This clearly shows that $S[p..p_\mathsf{c})u^l <_\mathsf{lex} S[i..i_\mathsf{c})$, so the new group of $p$ must be placed before the group of $i$.

Summing everything up, the group placements of new groups in line 12 of the algorithm ensure a correct new group order, so proposition (ii) holds after the iteration. □

Now, after having proved the correctness of the avalanche effect, we can make sure that Phase 1 of Algorithm 2 delivers an identical group division as that of the basic sorting principle.

**Theorem 4.1.2.** *Let $S$ be a nullterminated string of length $n$. When applying Algorithm 2 to $S$, the following propositions hold after Phase 1 is completed:*

- $\text{group}(i) = \text{group}(j) \Leftrightarrow S[i..\widehat{i}) =_\mathsf{lex} S[j..\widehat{j}) \quad \forall i, j \in [1 \ldots n]$
- $\text{group}(i) < \text{group}(j) \Leftrightarrow S[i..\widehat{i}) <_\mathsf{lex} S[j..\widehat{j}) \quad \forall i, j \in [1 \ldots n]$

*Proof.* Consider the point in time when the last group (\$) of $S$ is processed. Since this group is the lowest one, it will not change any further groups when being processed, and since \$ occurs only once at the end of $S$, the group order is correct after the start of the algorithm.

We show that $i_{\mathsf{c}} = \widehat{i}$ for all $i \in [1 \dots n]$ before the last group was processed, by which in combination with Lemma 4.1.1 (propositions (i) and (ii)) the theorem automatically follows. For $i = n$ the theorem already is correct, as shown above.

First, assume $\widehat{i} < i_{\mathsf{c}}$ for any $i \in [1 \dots n)$. By using proposition (iv) of Lemma 4.1.1, it follows that $\text{group}(i_{\mathsf{c}}) \leq \text{group}(i) < \text{group}(\widehat{i})$. Then, using proposition (ii) of Lemma 4.1.1, $S[i..i_{\mathsf{c}}) <_{\mathsf{lex}} S[\widehat{i}..\widehat{i}_{\mathsf{c}})$ must hold. Because $\text{group}(\widehat{i}) > \text{group}(i_{\mathsf{c}})$, $\widehat{i}_{\mathsf{c}} \leq i_{\mathsf{c}}$ must hold; otherwise, proposition (iv) of Lemma 4.1.1 would be harmed for $\widehat{i}$. This means that $S[i..i_{\mathsf{c}})$ cannot be a proper prefix of $S[\widehat{i}..\widehat{i}_{\mathsf{c}})$. Since $S[i..i_{\mathsf{c}}) <_{\mathsf{lex}} S[\widehat{i}..\widehat{i}_{\mathsf{c}})$, a $k < \widehat{i}_{\mathsf{c}} - \widehat{i}$ with $S[i..i+k) = S[\widehat{i}..\widehat{i}+k)$ and $S[i+k] < S[\widehat{i}+k]$ must exist. This further implies $S_i <_{\mathsf{lex}} S_{\widehat{i}}$ what leads to contradiction against Definition 3.0.1 of next lexicographically smaller suffixes.

So far, we know that $i_{\mathsf{c}} \leq \widehat{i}$ holds for all $i \in [1 \dots n)$. The next claim to be shown is that $i_{\mathsf{c}} = \widehat{i}$ for all $i \in [1 \dots n)$. The proof is done by induction on the distance $\widehat{i} - i$ of a suffix $S_i$.

**Induction Base:** Let $i \in [1 \dots n)$ be an index with $\widehat{i} - i = 1$.
Since $i < i_{\mathsf{c}} \leq \widehat{i}$ holds, clearly, $i_{\mathsf{c}} = \widehat{i}$.

**Induction Step:** Let $i \in [1 \dots n)$ be an index with $\widehat{i} - i > 1$.
Assume that $i_{\mathsf{c}} \neq \widehat{i}$, so $i_{\mathsf{c}} < \widehat{i}$. Using Definition 3.0.1 of next lexicographically smaller suffixes, $\widehat{i} <_{\mathsf{lex}} i <_{\mathsf{lex}} i_{\mathsf{c}}$ holds. Using the same definition, $i_{\mathsf{c}} >_{\mathsf{lex}} \widehat{i}$ implies $\widehat{i_{\mathsf{c}}} \leq \widehat{i}$.

Now we know that $i < i_{\mathsf{c}} < \widehat{i_{\mathsf{c}}} \leq \widehat{i}$. Since the distance between $i_{\mathsf{c}}$ and $\widehat{i_{\mathsf{c}}}$ is smaller than that of $i$ and $\widehat{i}$, we can apply the induction hypothesis, and obtain $i_{\mathsf{cc}} = \widehat{i_{\mathsf{c}}}$.

By our assumptions, $i <_{\mathsf{lex}} i_{\mathsf{c}}$ holds, so $S[i..i_{\mathsf{c}}) \leq_{\mathsf{lex}} S[i_{\mathsf{c}}..i_{\mathsf{cc}})$ follows. First, consider $S[i..i_{\mathsf{c}}) <_{\mathsf{lex}} S[i_{\mathsf{c}}..i_{\mathsf{cc}})$: Using proposition (ii) of Lemma 4.1.1, we obtain $\text{group}(i) < \text{group}(i_{\mathsf{c}})$. But, as (iv) of Lemma 4.1.1 states, $\text{group}(i) \geq \text{group}(i_{\mathsf{c}})$ should hold, contradiction.

Next, consider $S[i..i_{\mathsf{c}}) =_{\mathsf{lex}} S[i_{\mathsf{c}}..i_{\mathsf{cc}})$. Using $i_{\mathsf{cc}} = \widehat{i_{\mathsf{c}}}$ and the definition of next lexicographically smaller suffixes, $S_i = S[i..i_{\mathsf{c}})S_{i_{\mathsf{c}}} >_{\mathsf{lex}} S[i..i_{\mathsf{c}})S_{\widehat{i_{\mathsf{c}}}} = S_{i_{\mathsf{c}}}$ holds. Summarizing, $S_i <_{\mathsf{lex}} S_{i_{\mathsf{c}}}$ and $S_i >_{\mathsf{lex}} S_{i_{\mathsf{c}}}$ should hold, contradiction.
So clearly, our assumption was wrong, and $i_{\mathsf{c}} = \widehat{i}$ must hold. □

As expected, Phase 1 of Algorithm 2 sorts suffixes in groups just as desired. The remaining issue now is to prove that the whole algorithm works correctly, what will be addressed next.

**Theorem 4.1.3.** *Let $S$ be a nullterminated string of length $n$. Then, Algorithm 2 applied to the string $S$ computes the suffix array* $\mathsf{SA}$ *of $S$.*

*Proof.* Per induction on the iterations of the outer loop in Phase 2 (lines 17 to 28). The induction hypothesis states that before the $i$-th iteration

(i) $\mathsf{SA}[i] = j$ for the lexicographically $i$-th smallest suffix $S_j$.

(ii) For any $j \in [1 \dots n)$, $S_{\widehat{j}} <_{\mathsf{lex}} S_{\mathsf{SA}[i]} \Leftrightarrow S_j$ is correctly placed in the suffix array.[2]

(iii) The group order is *consistent*:
$\text{group}(j) < \text{group}(k) \Rightarrow S_j <_{\mathsf{lex}} S_k \ \forall j, k \in [1 \dots n]$.

[2] $\mathsf{SA}[k] = j \Leftrightarrow S_j$ is the lexicographically $k$-th smallest suffix.

(iv) Correctly placed suffixes belong to their own group:
If $\mathsf{SA}[k] = j$ for any $j, k \in [1 \dots n]$, then $|\operatorname{group}(j)| = 1$.

**Induction Base:** Before the first iteration, line 16 sets $\mathsf{SA}[1] = n$. Because of the definition of the sentinel character \$ this is correct, so (i) is fulfilled. Since $S_n$ is the lexicographically smallest suffix, there exists no suffix $S_j$ with $S_{\widehat{j}} <_{\mathsf{lex}} S_n$. Initially, the suffix array is filled with `nil`'s except for the first position, thus (ii) is correct, too. Also, because of the suffix grouping after Phase 1 (Theorem 4.1.2) and Theorem 3.1.1 of page 10, (iii) and (iv) hold.

**Induction Step:** Consider Algorithm 2 in the $i$-th iteration. We will first show (ii), (iii) and (iv), and then use this result to show (i).

So first, we need to show that after the $i$-th iteration all suffixes $S_j$ with $S_{\widehat{j}} \leq_{\mathsf{lex}} S_{\mathsf{SA}[i]}$ are correctly placed in the suffix array. By induction hypothesis we know that all suffixes $S_j$ with $S_{\widehat{j}} <_{\mathsf{lex}} \mathsf{SA}[i]$ are placed correct already, so it's sufficient to show that all suffixes $S_j$ with $\widehat{j} = \mathsf{SA}[i]$ are placed correctly during the $i$-th iteration.

Within the $i$-th iteration, the algorithm iterates over indices $\mathsf{SA}[i-1], \operatorname{prev}(\mathsf{SA}[i-1]), \operatorname{prev}(\operatorname{prev}(\mathsf{SA}[i-1])), \dots$, until index 0 or an index $j$ with $\mathsf{SA}[|\{\ s \in [1 \dots n] \mid \operatorname{group}(s) < \operatorname{group}(j)\ \}| + 1] \neq \texttt{nil}$ is reached, lines 17 to 28. First, let's discuss the second loop termination criteria. Let $sr \coloneqq |\{\ s \in [1 \dots n] \mid \operatorname{group}(s) < \operatorname{group}(j)\ \}|$. Then the observation is the following:

$$j \text{ is placed into the suffix array already } \Leftrightarrow \mathsf{SA}[sr+1] \neq \texttt{nil} \tag{4.3}$$

If $j$ has been placed in the suffix array already, using hypothesis (iii) and (iv), $sr$ is the number of lexicographically smaller suffixes of $S_j$. Because of the correct placement of $j$, $\mathsf{SA}[sr+1] = j$ must hold, so especially $\mathsf{SA}[sr+1] \neq \texttt{nil}$ holds. Now, consider $\mathsf{SA}[sr+1] \neq \texttt{nil}$. By hypothesis (iv), $|\operatorname{group}(\mathsf{SA}[sr+1])| = 1$ holds. Consequently, $|\{\ s \in [1 \dots n] \mid \operatorname{group}(s) < \operatorname{group}(\mathsf{SA}[sr+1])\ \}| = |\{\ s \in [1 \dots n] \mid \operatorname{group}(s) < \operatorname{group}(j)\ \}|$, so $\operatorname{group}(\mathsf{SA}[sr+1]) = \operatorname{group}(j)$. Since $\mathsf{SA}[sr+1]$ belongs to its own group, $\mathsf{SA}[sr+1] = j$, so $j$ is placed into the suffix array already.

Using observation (4.3) and Definition 4.1.1 of prev pointer chains from page 24, the behaviour of the algorithm in the $i$-th iteration can be described as following: Let $k$ be the smallest number such that $\pi^k(\mathsf{SA}[i]-1)$ is not contained in the suffix array. Then, the algorithm iterates the set $\mathcal{J} \coloneqq \{\ \pi^l(\mathsf{SA}[i]-1) \mid l \in [0 \dots k)\ \}$, places each index $j \in \mathcal{J}$ to the suffix array, and creates a new group for each index. Thus, to show that all suffixes $S_j$ with $\widehat{j} = \mathsf{SA}[i]$ are placed correctly to the suffix array, we need to show that $\mathcal{M} \coloneqq \{\ s \in [1 \dots n] \mid \widehat{s} = \mathsf{SA}[i]\ \} = \mathcal{J}$, as well as showing that each placement is correct.

We start with the first part, namely we show that $j \in \mathcal{J} \Leftrightarrow j \in \mathcal{M}$. For the forward direction, the proof is done per induction over the prev pointer chain of $\mathsf{SA}[i]-1$: As base case, consider $j \coloneqq \mathsf{SA}[i]-1$ is not placed into the suffix array. Then, using hypothesis (i) in the $i$-th iteration, we know that $S_j >_{\mathsf{lex}} S_{\mathsf{SA}[i]}$. Consequently, using the definition of next lexicographically smaller suffixes, $\widehat{j} = \mathsf{SA}[i]$.
For the induction step, for some $l > 0$, let $j \coloneqq \pi^l(\mathsf{SA}[i]-1)$ and $k \coloneqq \pi^{l-1}(\mathsf{SA}[i]-1)$ be indices such that $j$ is not contained in the suffix array, and $\widehat{k} = \mathsf{SA}[i]$ holds. Since $\operatorname{prev}(k) = j$, $j = \min\{\ s \in [1 \dots k) \mid \operatorname{group}(s) < \operatorname{group}(k)\ \}$, see line 5 of the algorithm,

as well as the argumentation in Lemma 4.1.1. Now, using the prev pointer definition and the consistent group order (hypothesis (iii)), group$(j) <$ group$(q)$ implies $S_j <_{\mathsf{lex}} S_q$ for all $j < q \leq k$. Also, using the definition of next lexicographically smaller suffixes for $S_k$, $S_j <_{\mathsf{lex}} S_k <_{\mathsf{lex}} S_q$ for all $k < q < \widehat{k}$. Consequently, $\widehat{j} \geq \widehat{k} = \mathsf{SA}[i]$ must hold. Since $j$ is not contained in the suffix array, using hypothesis (i) in the $i$-th iteration, $S_j >_{\mathsf{lex}} S_{\mathsf{SA}[i]}$, so clearly, $\widehat{j} = \mathsf{SA}[i]$.

For the backward direction, let $j$ be an index such that $j \notin \mathcal{J}$. If $j \geq \mathsf{SA}[i]$, clearly $\widehat{j} \neq \mathsf{SA}[i]$ holds by the definition of next lexicographically smaller suffixes. Next, consider the case that $j$ is placed between an index $k$ and its prev pointer, i.e. prev$(k) < j < k$, and $\widehat{k} = \mathsf{SA}[i]$ holds. Using the definition of prev pointers, group$(j) \geq$ group$(k)$ holds. Now, using proposition (iv) of Lemma 4.1.1 after Phase 1, $j_{\mathsf{c}} \leq k$ must hold; if $j_{\mathsf{c}} > k$, group$(j) <$ group$(k)$ must hold, contradiction. Since the end of the implicit context equals the next lexicographic smaller suffix (see Theorem 4.1.2), $\widehat{j} = j_{\mathsf{c}} \leq k < \mathsf{SA}[i]$ holds, so $\widehat{j} \neq \mathsf{SA}[i]$.
For the last missing case, let $S_j$ be a suffix already contained in the suffix array, such that prev$(k) = j$ for any $k \in \mathcal{J}$. We need to show that $\widehat{q} \neq \mathsf{SA}[i]$ for any $q \in [1 \dots j]$. If $q = j$, $j$ is contained in the suffix array already. Using hypothesis (ii), $S_{\widehat{j}} <_{\mathsf{lex}} S_{\mathsf{SA}[i]}$ holds, so $\widehat{j} \neq \mathsf{SA}[i]$. Now, assume for any $q$ with $1 \leq q < j$ that $\widehat{q} = \mathsf{SA}[i]$. Since $q < j < \mathsf{SA}[i]$, using the definition of next lexicographically smaller suffixes for $q$, $S_q <_{\mathsf{lex}} S_j$ must hold. Using the same definition, $S_{\mathsf{SA}[i]} = S_{\widehat{q}} <_{\mathsf{lex}} S_q <_{\mathsf{lex}} S_j$ holds. Since $j < \widehat{q}$ holds by precondition, this means that $\widehat{j} \leq \mathsf{SA}[i]$ and $S_{\widehat{j}} \geq_{\mathsf{lex}} S_{\mathsf{SA}[i]}$ hold. Thus, by hypothesis (ii), the suffix $S_j$ cannot be contained in the suffix array, contradiction.

Both directions show that $j \in \mathcal{J} \Leftrightarrow j \in \mathcal{M}$. The missing part is to ensure correct placement of any $j \in \mathcal{J}$ into the suffix array. Therefore, consider $j$ to be an index with $\widehat{j} = \mathsf{SA}[i]$. Because of the consistent group order (hypothesis (iv)), all suffixes placed in lower groups are lexicographically than $S_j$. Now, assume that another suffix $S_k <_{\mathsf{lex}} S_j$ with group$(k) =$ group$(j)$ exists. Since $j$ and $k$ belong to the same group, using Lemma 4.1.2, $S[j..\widehat{j}) = S[k..\widehat{k})$ must hold. Thus, $S_{\widehat{k}} <_{\mathsf{lex}} S_{\widehat{j}}$ must hold. In this case, using hypothesis (ii), $k$ must have been placed in the suffix array already. Since $k$ is placed in the suffix array already, it must belong to its own group, as hypothesis (iv) states. Consequently, group$(k) \neq$ group$(j)$, so no such suffix can exist, and $S_j$ must be the lexicographically minimal element of its group. Thus, line 20 computes the number of lexicographically smaller suffixes of $S_j$, and line 24 places $j$ at the correct position into the suffix array. Furthermore, since line 25 places $j$ into its own group, hypothesis (iv) is correct after the $i$-th iteration. Finally, since $S_j$ is the lexicographically minimal suffix of it's old group, the placement of the new group as immediate predecessor of $j$'s old group ensures a consistent group order, so hypothesis (iii) is correct after the $i$-th iteration.

So far, we proved hypotheses (ii), (iii) and (iv). Now, it remains to show that after the $i$-th iteration the lexicographically $i+1$-th smallest suffix $S_j$ is placed in $\mathsf{SA}$, i.e. $\mathsf{SA}[i+1] = j$. Since $S_j$ is the lexicographically $i+1$-th lowest suffix in $S$, $S_{\widehat{j}} \leq_{\mathsf{lex}} S_{\mathsf{SA}[i]}$ must hold. Now, using hypothesis (i) which is proved already, we know that all suffixes $S_k$ with $S_{\widehat{k}} = S_{\widehat{j}}$ are correctly placed in the suffix array. As $S_j$ belongs to those suffixes, $\mathsf{SA}[i+1] = j$ must hold, thus hypothesis (i) is shown.

Since all suffixes are placed correctly to the suffix array, and entries are not changed (line 21), Algorithm 2 computes the correct and entire suffix array $\mathsf{SA}$ of $S$. □

To be honest, the correctness proof of Algorithm 2 is quite hard. On the other hand, the algorithm itself is relatively easy to understand—in my opinion at least. It seems a bit as if the complexity degree of efficient suffix array construction must been split up between an algorithm and its proof, forming a balance of complexity between all suffix array construction algorithms of same asymptotic runtime, but that's just a marginal note I was thinking of while proving correctness.

However, we've seen the algorithm and its correctness, but one part is missing: asymptotic runtime. So, within the next section, the time complexity of the algorithm will be discussed.

## 4.2 Runtime

After proving correctness of the algorithm, next issue will be to prove the asymptotic linear runtime of the algorithm, together with linear space consumption. Since the algorithm was described in a rather rough way yet, we need to get a bit more technical, but keeping everything as simple as possible. A more appropriate implementation for real world usage can be found within the next chapter.

First, recall Phase 1 of Algorithm 2. Main tasks were to build initial groups, iterate them in descending group order, compute previous smaller suffixes and rearranging them. In order to explain all necessary steps, Algorithm Excerpt 2 shows the performed instructions during Phase 1.

**Algorithm Excerpt 2** Phase 1 of Algorithm 2, page 23.

*Phase 1: sort suffixes into groups*

1: order all suffixes of $S$ into groups according to their first character:
Let $S_i$ and $S_j$ be two suffixes. Then, $\text{group}(i) = \text{group}(j) \Leftrightarrow S[i] = S[j]$.
2: order the suffix groups: Let $\mathcal{G}_1$ be a suffix group with group prefix character $u$, $\mathcal{G}_2$ be a suffix group with group prefix character $v$. Then, $\mathcal{G}_1 < \mathcal{G}_2$ if $u < v$.
3: **for each** group $\mathcal{G}$ in descending group order **do**
4: **for each** $i \in \mathcal{G}$ **do**
5: $\text{prev}(i) \leftarrow \max(\{\ j \in [1 \dots i-1] \mid \text{group}(j) < \text{group}(i)\ \} \cup \{0\})$
6: **end for**
7: let $\mathcal{P}$ be the set of previous suffixes from $\mathcal{G}$,
$\mathcal{P} \coloneqq \{\ j \in [1 \dots n] \mid \text{prev}(i) = j \text{ for any } i \in \mathcal{G}\ \}$.
8: split $\mathcal{P}$ into $k$ subsets $\mathcal{P}_1, \dots, P_k$ such that a subset $P_l$ contains suffixes whose number of prev pointers from $\mathcal{G}$ pointing to them is equal to $l$, i.e. $i \in \mathcal{P}_l \Leftrightarrow |\{\ j \in \mathcal{G} \mid \text{prev}(j) = i\ \}| = l$.
9: **for** $l = k$ **down to** 1 **do**
10: split $\mathcal{P}_l$ into $m$ subsets $\mathcal{P}_{l_1}, \dots, \mathcal{P}_{l_m}$ such that suffixes of same group are gathered in the same subset.
11: **for** $q = 1$ **up to** $m$ **do**
12: remove suffixes of $\mathcal{P}_{l_q}$ from their group and put them into a new group placed as immediate successor of their old group.
13: **end for**
14: **end for**
15: **end for**

First thing that has to be done is to explain a working set of needed data structures. Five arrays of size $n$ will be used for this:

- SA contains suffix starting positions, ordered according to the current group order.
- ISA is the inverse permutation of SA, to be able to detect the position of a certain suffix in SA.
- GSIZE contains the sizes of all groups. Group sizes are ordered according to the group order, so GSIZE has same order as SA. GSIZE contains the size of each group only once at the beginning of the group, followed by zeros until the beginning of the next group.
- GLINK stores pointers from suffixes to their groups. All entries point at the beginning of a group, at the same position where GSIZE contains the size of the group.
- PREV is used to store prev pointers computed during Phase 1, see line 5 of Algorithm Excerpt 2. All entries initially are set to `nil`, to detect if a prev pointer already exists.

An example of the data structure setup can be found in Figure 4.1. The initial setup of those arrays can be performed in linear time by using a technique called *bucket sort* and further iterations using a character count table. Thus, lines 1 and 2 require $O(n)$ time.

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $S[i]$ | g | r | a | i | n | d | r | a | i | n | i | n | g | $ |
| GSIZE$[i]$ | 1 | 2 | 0 | 1 | 2 | 0 | 3 | 0 | 0 | 3 | 0 | 0 | 2 | 0 |
| SA$[i]$ | 14 | 3 | 8 | 6 | 1 | 13 | 4 | 9 | 11 | 5 | 10 | 12 | 2 | 7 |
| GLINK$[i]$ | 5 | 13 | 2 | 7 | 10 | 4 | 13 | 2 | 7 | 10 | 7 | 10 | 5 | 1 |
| ISA$[i]$ | 5 | 13 | 2 | 7 | 10 | 4 | 14 | 3 | 8 | 11 | 9 | 12 | 6 | 1 |

Figure 4.1: Initial data structure setup after line 2 of Phase 1, applied to the string $S =$ `graindraining`$. Prev pointers are not listed since all entries initially are set to `nil`, some (but not all) GLINK pointers are displayed for better illustration.

The first problem that has to be solved is to process the groups in descending group order, line 3. Assume we got two variables $gs$ and $ge$, pointing to start and end of a group. To get to the next lower group, we can set $ge \leftarrow gs - 1$ and $gs \leftarrow \mathsf{GLINK}[\mathsf{SA}[gs - 1]]$. Doing group iteration in this way, we trivially need $O(n)$ steps to process all groups in descending group order. Also, the suffixes of the processed group can be accessed by iterating $\mathsf{SA}[gs..ge]$. Initially, $gs$ can be set to $n + 1$, to start with the highest group.

Next thing to be handled is the computation of prev pointers, line 5 of Algorithm Excerpt 2. As already mentioned, we use a technique called *pointer jumping* for this purpose: From an index $s$, we start with its previous index $s - 1$. If $s - 1$ belongs to

a lower group, we're done already. If $s-1$ belongs to a higher group, its prev pointer must have been computed already. Since $s-1$ is placed in a higher group, all suffixes between $s-1$ and $\mathsf{PREV}[s-1]$ belong to higher groups than that of $s$, so we can use $\mathsf{PREV}[s-1]$ to jump above all of those suffixes. After jumping, the procedure will be repeated, until an index with a lower or equal group is reached, see Algorithm 3.

Let us shortly ignore the special case when Algorithm 3 returns a suffix placed in the same group as $s$, we will handle this later on. If every call of function PREV-EQUAL would stop within the first iteration of its inner loop, overall prev pointer computation would require $O(n)$ work, since it is executed exactly once per suffix. So clearly, the question is how much additional iterations are performed by pointer jumping.

**Algorithm 3** Computation of a prev pointer for the index $s$ where $ge$ indicates the end of the group of $s$. Note that the returned index can belong to the same group as $s$.

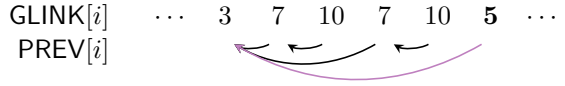
```
1:  function PREV-EQUAL(s, ge)
2:      p ← s − 1
3:      while p > 0 do
4:          if ISA[p] ≤ ge then                    ▷ group(p) ≤ group(s)
5:              return p
6:          end if
7:          p ← PREV[p]                            ▷ PREV[p] must exist already
8:      end while
9:      return 0
10: end function
```

To answer the question, we first will show that prev pointers cannot cross: Let $s, \tilde{s}$ be two integers in range $[1 \dots n]$ with $\mathsf{PREV}[s] < \mathsf{PREV}[\tilde{s}] < s < \tilde{s}$. Applying the definition of prev pointers (line 5 of Algorithm Excerpt 2) to $\tilde{s}$, $\text{group}(\mathsf{PREV}[\tilde{s}]) < \text{group}(\tilde{s}) \leq \text{group}(s)$ must hold. Also, by applying the definition to $s$, $\text{group}(\mathsf{PREV}[\tilde{s}]) \geq \text{group}(s)$ holds, contradiction, prev pointers cannot cross.

After the computation of a prev pointer $\mathsf{PREV}[s]$ in Algorithm 3, all used pointers from the pointer jumping technique are overlaid by the new pointer, see Figure 4.2 for an example. Assume that the pointers used for pointer jumping will be used again, in a later step. Clearly, all prev pointers for indices between $s$ and $\mathsf{PREV}[s]$ already are computed, because of the descending group order processing. Since a prev pointer is computed only once per suffix, and Phase 1 processes groups in descending group order, an index $\tilde{s}$ with $\mathsf{PREV}[s] < \mathsf{PREV}[\tilde{s}] < s < \tilde{s}$ must exist; otherwise, the indices between $\mathsf{PREV}[s]$ and $s$ cannot be accessed. But prev pointers cannot cross, so no pointer used by the pointer jumping technique is used more than once. Since at most $n$ prev pointers are computed, at most $n$ pointers can be used by pointer jumping, so the overall prev pointer computation requires $O(n)$ work.

Now, let's come back to the special case when Algorithm 3 returns an index that belongs to the same group as its input index, i.e. $\text{group}(s) = \text{group}(\text{PREV-EQUAL}(s, ge))$ for some $s \in [1 \dots n]$. To solve this case, we need to build some additional instructions: Let $s$ be an index that belongs to the currently processed group. If the prev pointer of $s$ is computed already, we proceed with the next index of the currently processed group. Otherwise, we compute $p \leftarrow \text{PREV-EQUAL}(s, ge)$. If $p$ belongs to a lower ordered group, we set $\mathsf{PREV}[s] \leftarrow p$; if $p$ belongs to the same group, $s$ is added to a list $\mathcal{L}$, and


(1) Computation of a prev pointer using pointer jumping.




(2) All pointers used by pointer jumping are overlaid by the new prev pointer.

Figure 4.2: Prev pointer computation using pointer jumping.

the procedure is repeated for $s \leftarrow p$. At some point, the prev pointer for $s$ is correctly computed. All remaining indices $l \in \mathcal{L}$ however still require a prev pointer. Since the prev pointer of $s$ is correct already, caused by function PREV-EQUAL, $\text{group}(j) \geq \text{group}(l)$ for all $s \leq j < l$ holds for all $l \in \mathcal{L}$. The indices of $\mathcal{L}$ belong to the same group as $s$, so we set $\mathsf{PREV}[l] \leftarrow \mathsf{PREV}[s]$ for all $l \in \mathcal{L}$, resulting in correct prev pointers for all indices of $\mathcal{L}$. Thus, Algorithm 4 shows how prev pointers of the currently processed group can be computed.

**Algorithm 4** Prev pointer computation of the processed group, lines 4 to 6 of Algorithm Excerpt 2. Indices of the current group are contained in $\mathsf{SA}[gs..ge]$.

```
1:  for i ← gs up to ge do
2:      s ← SA[i]
3:      initialize an empty list L
4:      while PREV[s] = nil do
5:          p ←PREV-EQUAL(s, ge)
6:          if p = 0 or ISA[p] < gs then          ▷ p points to a suffix in a lower group
7:              PREV[s] ← p
8:          else                                  ▷ p points to a suffix in same group
9:              add s to list L
10:             s ← p
11:         end if
12:     end while
13:     for each l ∈ L do                         ▷ copy prev pointer from s to all stored indices
14:         PREV[l] ← PREV[s]
15:     end for
16: end for
```

This extra computation requires at most $O(|\mathcal{G}|)$ extra time (where $\mathcal{G}$ is the processed group), and at most $O(n)$ space for the list. Therefore, the prev pointer computation still is doable in $O(n)$ time, since each group is processed exactly once.

After handling prev pointer computation, next task is to compute the set $\mathcal{P}$ of previous suffixes, and split it into subsets $\mathcal{P}_1, \ldots, \mathcal{P}_k$, such that each subset $\mathcal{P}_l$ consists of suffixes $S_i$ with exactly $l$ indices of $\mathcal{G}$ pointing onto $i$, i.e. $i \in \mathcal{P}_l \Leftrightarrow |\{\ s \in \mathcal{G} \mid \text{prev}(s) = i\}| = l$, see lines 7 to 8 of Algorithm Excerpt 2. An extra array $\mathsf{PC}$ of size $n$ is used for this

purpose. Initially, all entries of PC are set to zero. Then, after computing prev pointers in Algorithm 4, for all $s \in \mathcal{G}$, PC[PREV[$s$]] is incremented. During this loop, the set $\mathcal{P}$ can be computed by checking if PC[PREV[$s$]] $= 0$ holds: if true, PREV[$s$] is visited the first time and can be appended to a list; otherwise, PREV[$s$] is contained in the list already. Also, after the loop, for each $p \in \mathcal{P}$, PC[$p$] contains the count of prev pointers of $\mathcal{G}$ pointing onto $p$, so for the second split, $p$ belongs to the set $\mathcal{P}_{\mathsf{PC}[p]}$.

The second split from $\mathcal{P}$ into subsets $\mathcal{P}_1, \ldots, \mathcal{P}_k$ can be performed like the following: While $\mathcal{P}$ is not empty, for each $p \in \mathcal{P}$, decrement PC[$p$]. If PC[$p$] $= 0$ within the $l$-th outer iteration, remove $p$ from $\mathcal{P}$, and add it to the set $\mathcal{P}_l$. This way, the sets $\mathcal{P}_1, \ldots, \mathcal{P}_k$ are computed in increasing order, and all entries of PC are set to zero, so it can be reused for the next split operation. The subsets $\mathcal{P}_l$ are stored in the SA-interval of the processed group: define $\mathrm{pls}(l) := ge + 1 - \sum_{i=0}^{l} |\mathcal{P}_i|$. Then the set $\mathcal{P}_1$ is stored to SA[pls(1).. pls(0) $-$ 1], the set $\mathcal{P}_2$ is stored to SA[pls(2).. pls(1) $-$ 1], and so on. This way, empty subsets are ignored, and the sets are stored in decreasing order. Additionally, to know the size of each set, we set GSIZE[pls($l$)] $\leftarrow |\mathcal{P}_l|$ for each non-empty set. Because the indices of $\mathcal{G}$ aren't used for further instructions, and $|\mathcal{P}| \leq |\mathcal{G}|$ holds, no side effects occur during the storage. Algorithm 5 shows the code for splitting previous suffixes.

**Algorithm 5** Computation of set $\mathcal{P}$ and subsets $\mathcal{P}_1, \ldots, \mathcal{P}_k$, lines 7 to 8 of Algorithm Excerpt 2. The sets $\mathcal{P}_1, \ldots, \mathcal{P}_k$ are stored into the suffix array interval of the group, from left to right in decreasing order.

```
 1: initialize an empty list P
 2: for i ← gs up to ge do                    ▷ compute set P and count prev pointers
 3:     p ← PREV[SA[i]]
 4:     if p > 0 then
 5:         if PC[p] = 0 then                                  ▷ p was not counted yet
 6:             add p to P
 7:         end if
 8:         PC[p] ← PC[p] + 1                  ▷ count pointers from current group to p
 9:     end if
10: end for
11: pls ← ge + 1                                ▷ start of set P_l in the suffix array
12: ple ← ge                                    ▷ end of set P_l in the suffix array
13: while P is not empty do      ▷ within the l-th iteration, the set P_l is computed
14:     for each p ∈ P do
15:         PC[p] ← PC[p] − 1
16:         if PC[p] = 0 then
17:             remove p from P
18:             pls ← pls − 1
19:             SA[pls] ← p                                ▷ store p to set P_l
20:         end if
21:     end for
22:     if pls ≤ ple then                                  ▷ set P_l contains elements
23:         GSIZE[pls] ← ple − pls + 1                     ▷ store size of P_l
24:         ple ← pls − 1                                  ▷ prepare ple for next set
25:     end if
26: end while
```

The first loop of Algorithm 5 requires $\mathcal{O}(|\mathcal{G}|)$ time, since all indices of $\mathcal{G}$ are iterated. Also, the second loop requires $\mathcal{O}(|\mathcal{G}|)$ time: its inner loop overall requires as much iterations as prev pointers from $\mathcal{G}$ exist, so consequently, at most $\mathcal{O}(|\mathcal{G}|)$ iterations are performed.

So, let's move on to the final step in Phase 1, the rearrangements from lines 9 to 14 in Algorithm Excerpt 2. By the previous splits, we're able to iterate subsets $\mathcal{P}_l$ in descending order. It actually turns out that the additional split of a subset $\mathcal{P}_l$ from line 10 is not required to rearrange suffixes: First, for all $p \in \mathcal{P}_l$, decrement $\mathsf{GSIZE}[\mathsf{GLINK}[p]]$, and exchange $\mathsf{SA}[\mathsf{ISA}[p]]$ with the rightmost suffix of its group, $\mathsf{SA}[\mathsf{GLINK}[p]+\mathsf{GSIZE}[\mathsf{GLINK}[p]]]$. Also, update the new ISA values, so ISA stays the correct inverse permutation of SA. This way, suffixes in $\mathcal{P}_l$ are removed from their groups, because they'll be placed at the back of their groups, and the sizes of the groups don't cover them any more. Remaining task is to set up GLINK and GSIZE, so the new groups are correctly captured. First, for all $p \in \mathcal{P}_l$, set $\mathsf{GLINK}[p] \leftarrow \mathsf{GLINK}[p] + \mathsf{GSIZE}[\mathsf{GLINK}[p]]$. Because of the decrements of GSIZE in the previous loop, GLINK now correctly points to the group beginnings of the new groups. Last step is to increment $\mathsf{GSIZE}[\mathsf{GLINK}[p]]$ for all $p \in \mathcal{P}_l$, so the sizes of the new groups are set up correctly. Algorithm 6 shows the full code for rearrangements, an example can be found in Figure 4.3.

**Algorithm 6** Suffix rearrangements of Phase 1, lines 9 to 14 of Algorithm Excerpt 2. Variables *pls* and *ple* are the same as in Algorithm 5, and initially contain the same values as contained after the termination of Algorithm 5.

```
1: while pls ≤ ge do                                  ▷ iterate subsets in descending order
2:     ple ← pls + GSIZE[pls]                         ▷ SA[pls..ple] contains all suffixes of set P_l
3:     for i ← pls up to ple do                       ▷ decrement group sizes of previous suffixes
4:         p ← SA[i]
5:         sr ← GLINK[p]
6:         GSIZE[sr] ← GSIZE[sr] − 1
7:         sr ← sr + GSIZE[sr]
8:         s ← SA[sr]          ▷ move p to back of its group by exchange with last suffix
9:         pr ← ISA[p]
10:        SA[pr] ← s      ISA[s] ← pr
11:        SA[sr] ← p      ISA[p] ← sr
12:    end for
13:    for i ← pls up to ple do                       ▷ set new GLINK for rearranged suffixes
14:        p ← SA[i]
15:        pgs ← GLINK[p]
16:        pgs ← pgs + GSIZE[pgs]
17:        GLINK[p] ← pgs
18:    end for
19:    for i ← pls up to ple do                       ▷ set GSIZE for new groups
20:        p ← SA[i]
21:        pgs ← GLINK[p]
22:        GSIZE[pgs] ← GSIZE[pgs] + 1
23:    end for
24:    pls ← ple + 1
25: end while
```

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | $-1$ | $-1$ | | | | |
| GSIZE$[i]$ | 1 | 2 | 0 | 1 | 2 | 0 | 3 | 0 | $\cdots$ |
| SA$[i]$ | 14 | 3 | 8 | **6** | **1** | 13 | 4 | 9 | $\cdots$ |
| GLINK$[i]$ | **5** | 13 | 2 | 7 | 10 | **4** | 13 | 2 | $\cdots$ |

(1) for each $p \in \mathcal{P}_i$, move $p$ to the back of its group and decrement group size.

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | |
|---|---|---|---|---|---|---|---|---|---|
| GSIZE$[i]$ | 1 | 2 | 0 | 0 | 1 | 0 | 3 | 0 | $\cdots$ |
| GLINK$[i]$ | **5** | 13 | 2 | 7 | 10 | **4** | 13 | 2 | $\cdots$ |

(2) for each $p \in \mathcal{P}_i$, add gsize of old group to GLINK$[p]$.

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | $+1$ | | $+1$ | | | |
| GSIZE$[i]$ | 1 | 2 | 0 | 0 | 1 | 0 | 3 | 0 | $\cdots$ |
| GLINK$[i]$ | **6** | 13 | 2 | 7 | 10 | **4** | 13 | 2 | $\cdots$ |

(3) for each $p \in \mathcal{P}_i$, increment group size of new group.

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | |
|---|---|---|---|---|---|---|---|---|---|
| GSIZE$[i]$ | 1 | 2 | 0 | 1 | 1 | 1 | 3 | 0 | $\cdots$ |
| SA$[i]$ | 14 | 3 | 8 | **6** | 13 | **1** | 4 | 9 | $\cdots$ |
| GLINK$[i]$ | **6** | 13 | 2 | 7 | 10 | **4** | 13 | 2 | $\cdots$ |

(4) Result after the rearrangements. All suffixes of set $\mathcal{P}_l$ were rearranged in new groups, placed as immediate successors of their old groups.

Figure 4.3: Suffix rearrangements for the set $\mathcal{P}_l = \{1, 6\}$. Items of $\mathcal{P}_l$ are marked bold.

Since each subset $\mathcal{P}_l$ is iterated three times, Algorithm 6 requires $O(|\mathcal{P}|) = O(|\mathcal{G}|)$ asymptotic time.

Now, we're almost done with Phase 1. The last bit I want to mention here is a preparation step for Phase 2. Recall that $gs$ and $ge$ were the indices of the current group's start and end in SA. After the rearrangements took place, SA$[ge]$ will be set to $gs$. This last index of the group later will perform as a group counter. Also, to ensure each suffix has access to the group counter, we set ISA$[s] \leftarrow ge$ for all indices $s$ of the processed group. This step has to be performed before the computation of the set $\mathcal{P}$ in Algorithm 5, since Algorithm 5 overwrites the set indices with prev pointers. Note that this preparations don't cause side effects: Each group is processed only once, so SA can be overwritten without problems. Also, after the preparation, $gs \leq$ ISA$[s] \leq ge$ holds

for all indices $s$ of the processed group, so we can still determine if a suffix belongs to an already processed group.

Before Phase 2 will be discussed, let's first recall the performed steps of Phase 2, see Algorithm Excerpt 3. The only instructions not directly implementable are those from lines 20 to 25.

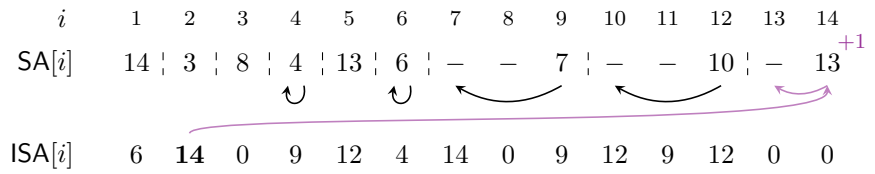
**Algorithm Excerpt 3** Phase 2 of Algorithm 2, page 24.

```
    Phase 2: construct suffix array from groups
16: SA[1] ← n
17: for i = 1 up to n do
18:     j ← SA[i] − 1
19:     while j ≠ 0 do
20:         let sr be the number of suffixes placed in lower groups,
             i.e. sr := |{ s ∈ [1 ... n] | group(s) < group(j) }|.
21:         if SA[sr + 1] ≠ nil then
22:             break
23:         end if
24:         SA[sr + 1] ← j
25:         remove j from its current group and put it in a new group
             placed as immediate predecessor of j's old group.
26:         j ← prev(j)
27:     end while
28: end for
```

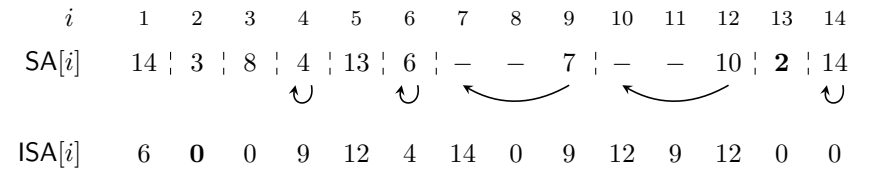
Let's start with the first point, checking if a suffix is contained in SA already, line 21. Since SA was modified during Phase 1, we cannot check if any SA - entry is `nil`. To solve this problem, we will use ISA: whenever an index $s$ is placed into SA, we set $\mathsf{ISA}[s] \leftarrow 0$. Then, to check if a suffix is placed into the suffix array already, we only need to compare its ISA-value with zero.

Next, we'll have a look at suffix rank computation and suffix placement in lines 20 and 25 of Algorithm Excerpt 3. Recall that each group was prepared in Phase 1, so for each suffix $S_j$, $\mathsf{ISA}[j]$ points to the end of its group in SA. Also, for every group, the end of a group contains a counter initially set to the start of the group in SA. To compute $sr$ in line 20, we follow this counter, i.e. $sr \leftarrow \mathsf{SA}[\mathsf{ISA}[j]]$. Then, to move a suffix to the front of its group, we set $\mathsf{SA}[sr] \leftarrow j$. To 'remove' the suffix $S_j$ from its old group, it's sufficient to increment the group counter $\mathsf{SA}[\mathsf{ISA}[j]]$. Note that the group counter must be incremented before the placement of $j$ in SA, since both positions can be equal. A full example for the placement of a suffix can be found in Figure 4.4, Algorithm 7 shows the full implementation of Phase 2.

Using this implementation of Phase 2, every described operation is supported in constant time. The overall time for Phase 2 is $O(n)$ plus the number of iterations of the inner loop. By the correctness of the algorithm (Theorem 4.1.3 from page 28) we know that within the $i$-th iteration of the outer loop, the inner loop will process all suffixes $j$ with $\widehat{j} = \mathsf{SA}[i]$. Since each suffix has exactly one next lexicographically smaller suffix, the inner loop iterates $n - 1$ suffixes, thus Phase 2 requires $O(n)$ time.




(1) let $sr \leftarrow \mathsf{SA}[\mathsf{ISA}[j]]$, then increment $\mathsf{SA}[\mathsf{ISA}[j]]$.




(2) set $\mathsf{SA}[sr]$ to $j$, and $\mathsf{ISA}[j]$ to zero.

Figure 4.4: Suffix placement for $j = 2$. Suffix items are marked bold, empty positions in SA are placed over arrows indicating the start of their groups.

**Algorithm 7** Implementation of Phase 2, see Algorithm Excerpt 3.

```
1:  SA[1] ← n
2:  for i = 1 up to n do
3:      j ← SA[i] − 1
4:      while j ≠ 0 do
5:          ge ← ISA[j]
6:          if ge = 0 then
7:              break
8:          end if
9:          sr ← SA[ge]
10:         SA[ge] ← SA[ge] + 1
11:         SA[sr] ← j
12:         ISA[j] ← 0
13:         j ← PREV[j]
14:     end while
15: end for
```

Now finally, we're able to show that the algorithm works in asymptotic optimal runtime and linear space.

**Theorem 4.2.1.** *Let $S$ be a nullterminated string of length $n$. Then, Algorithm 2 applied to the string $S$ computes the suffix array* SA *of $S$ in $O(n)$ time, with asymptotic linear word space.*

*Proof.* The initial group setup can be performed in $O(n)$. Also, computation of prev pointers requires $O(n)$ time. All remaining operations in the outer loop of Phase 1 require $O(|\mathcal{G}|)$ time, where $\mathcal{G}$ is the current processed group. Since each group is processed only once, overall time complexity for Phase 1 is $O(n)$. Phase 2 also can be performed

in $O(n)$ time, thus the overall time complexity of Algorithm 2 is $O(n)$. Each additional used array or list has length of at most $n$, so overall space complexity is $O(n)$ words. □

We've seen that the algorithm can be implemented with optimal asymptotic runtime and linear space. This is quite a nice result, but for real world applications, the algorithm needs to be tuned to consume as less space as possible and run as fast as possible. The next chapter will show some of such optimisations, resulting in a competitive linear–time suffix array construction algorithm.

# Chapter 5

# Implementation

After having presented the algorithm, this chapter's purpose is to describe an implementation suitable for real world usage. Within Section 4.2, a possible implementation was described already, but it has several downsides: A lot of additional data structures are required, and some algorithmic solutions seem a bit awkward. To fix these issues, the implementation of Section 4.2 will be modified.

First, the required data structure framework is described. It consists of four arrays of size $n$, exactly in the same manner as described in Section 4.2, page 32:

- SA contains suffix starting positions, ordered according to the current group order.
- ISA is the inverse permutation of SA, to be able to detect the position of a certain suffix in SA.
- GSIZE contains the sizes of all groups. Group sizes are ordered according to the group order, so GSIZE has same order as SA. GSIZE contains the size of each group only once at the beginning of the group, followed by zeros until the beginning of the next group.
- GLINK stores pointers from suffixes to their groups. All entries point at the beginning of a group, at the same position where GSIZE contains the size of the group.

In contrast to the implementation of Section 4.2, this will be all data structures we need.

Before we proceed on, let's shortly recapitulate the steps performed within the first phase of the algorithm. Pseudocode can be found in Algorithm Excerpt 2 on page 31, it won't be listed here again. Initially, suffixes were placed in groups according to their first character, groups themselves are sorted according to the rank of their representative character. This initial step can be done using *bucket sort*. Then, a loop processes groups in descending order. For each processed group $\mathcal{G}$, prev pointers are computed, the set $\mathcal{P}$ of previous suffixes is computed, and suffixes of $\mathcal{P}$ are split into subsets $\mathcal{P}_1, \ldots, \mathcal{P}_k$ according to the number of prev pointers of $\mathcal{G}$ pointing to them.[1] The last step is to process the subsets $\mathcal{P}_1, \ldots, \mathcal{P}_k$ in descending order, and rearranging suffixes of a subset $\mathcal{P}_l$ by placing all indices $p \in \mathcal{P}_l$ that belong to the same group into a new group, as immediate successor of the old group.

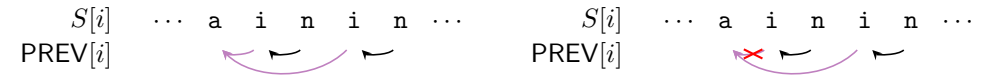
Processing groups in descending order works in the same way as in Section 4.2: Let $gs$ be the start of a group. To get to the next group, we set $ge$ to $gs - 1$ and then set

[1] A subset $\mathcal{P}_l$ consists of indices $p \in \mathcal{P}$, such that exactly $l$ prev pointers from $\mathcal{G}$ are pointing to $p$.

$gs \leftarrow \mathsf{GLINK}[ge]$. Now, the suffixes of the new group are contained in $\mathsf{SA}[gs..ge]$, and can be processed.

Next, let's move on to prev pointer computation. The first thing to be mentioned is the storage of prev pointers: no separate array is declared for this purpose. To solve this issue, prev pointers will be stored in the GLINK array: Since the GLINK array is used only for suffixes of groups that have not been processed yet, and rearrangements of suffixes only take place in unprocessed groups, no side effects occur. The only thing we need to be aware of is to use ISA to check if a group has been processed already: if $\mathsf{ISA}[s] > ge$, suffix $s$ belongs to a group that has been processed already.

Before proceeding, another optimisation shall be discussed. Consider the case when two prev pointers of suffixes from same group point to the same position, i.e. two indices $i, j \in \mathcal{G}$ with $i < j$ exist such that $\text{prev}(i) = \text{prev}(j)$, see Figure 5.1. In this case, the left prev pointer $\text{prev}(i)$ can be removed: since prev pointers do not cross, $\text{prev}(i)$ is overlaid by the right prev pointer $\text{prev}(j)$, thus it won't be used in the pointer jumping technique used for prev pointer computation. Additionally, in the second phase, the right prev pointer $\text{prev}(j)$ will be visited first[2], so the suffix $S_{\text{prev}(j)}$ will already be contained in the suffix array when $\text{prev}(i)$ is used. Since the algorithm stops in such cases, it doesn't matter whether the left prev pointer exists or not.




(1) Two prev pointers from same group point onto the same suffix.

(2) The left prev pointer is overlaid by the right one, and can be removed.

Figure 5.1: Prev pointer overlay by two pointers of same group.

Using this optimisation, prev pointer computation and the split of suffixes can be handled easier: First, for each suffix in the current group, compute PREV-EQUAL (see page 33). If the previous suffix belongs to the same group, mark it. Then, filter out all marked suffixes, and repeat the following procedure for all remaining suffix starting positions $s$: let $p$ be the prev pointer of $s$, $p = \mathsf{PREV}[s]$. if $p$ belongs to the same group as $s$, update the prev pointer of $s$ by using the prev pointer of its previous suffix $S_p$ ($\mathsf{PREV}[s] \leftarrow \mathsf{PREV}[p]$), and set $\mathsf{PREV}[p]$ to zero. Otherwise, add $p$ to a list, and remove $s$ from the remaining suffix starting positions. After the iteration of all remaining suffixes, store the computed list to a list of lists. By repeating this behaviour as long as suffixes remain, all prev pointers get computed correctly. Also, the generated lists correspond to the subsets $\mathcal{P}_1, \ldots, \mathcal{P}_k$. In essence, generating lists is not necessary, since the suffixes can be placed into the index storage of the currently processed group, namely $\mathsf{SA}[gs..ge]$: Each subset $\mathcal{P}_l$ is stored to an interval $\mathsf{SA}[ps..pe]$ with $gs \leq ps \leq pe \leq ge$, in the same manner as described in the previous chapter (Algorithm 5 on page 35). Algorithm 8 shows this prev pointer computation combined with the split of previous suffixes.

[2] Contexts of both suffixes $S_i$ and $S_j$ are the same. Since both prev pointers point to the same position, the context of the left suffix $S_i$ ends at the position of the right suffix $S_j$, i.e. $i_\mathsf{c} = j$. Since group $\mathcal{G}$ is processed at the moment, $i_\mathsf{c}$ won't be changed during the algorithm, so $\widehat{i} = i_\mathsf{c}$ holds by Theorem 4.1.2. This clearly shows that $\widehat{i} = j$, so the second phase will process $S_j$ before $S_i$.

**Algorithm 8** Prev pointer computation and split in the first phase of the algorithm, variables $gs$ and $ge$ contain start and end bounds of the current group. Every occurrence of PREV must be replaced with GLINK, but replacement is omitted for better readability.

```
 1: GSIZE[gs] ← 0                                          ▷ make place for markings
 2: for i ← ge down to gs do                               ▷ compute prev pointers
 3:     p ← PREV-EQUAL(SA[i], ge)                          ▷ Algorithm 3 on page 33
 4:     if p > 0 and ISA[p] ≥ gs then                      ▷ p belongs to current group
 5:         GSIZE[ISA[p]] ← 1                              ▷ mark p
 6:     end if
 7:     PREV[SA[i]] ← p                                    ▷ store pointer
 8: end for

 9: pe ← gs
10: for i ← gs up to ge do                                 ▷ move unmarked suffixes to front
11:     if GSIZE[i] ≠ 1 then
12:         SA[pe] ← SA[i]
13:         pe ← pe + 1
14:     end if
15:     ISA[SA[i]] ← ge                                    ▷ preparation for the second phase
16: end for

17: ps ← gs
18: pe ← pe + 1
19: l ← 0
20: repeat                                                 ▷ compute final pointers and split suffixes
21:     i ← pe − 1
22:     tmp ← pe
23:     while i ≥ ps do
24:         p ← PREV[SA[i]]
25:         if p > 0 then
26:             if ISA[p] < gs then                        ▷ p is in an other group
27:                 pe ← pe − 1
28:                 SA[i] ← SA[pe]
29:                 SA[pe] ← p                             ▷ push pointer to back
30:             else                                       ▷ p is in current group
31:                 PREV[SA[i]] ← PREV[p]                  ▷ copy pointer
32:                 PREV[p] ← 0          ▷ clear pointer of p, won't be used any more
33:             end if
34:             i ← i − 1
35:         else                                           ▷ p points to nothing, remove it
36:             SA[i] ← SA[ps]
37:             ps ← ps + 1
38:         end if
39:     end while
40:     if pe < tmp then          ▷ at least one prev pointer was pushed to back
41:         GSIZE[pe] ← tmp − pe                           ▷ store number of pointers
42:         l ← l + 1                                      ▷ and update number of subsets
43:     end if
44: until ps = pe
```

The rearrangements can be performed in exactly the same way as described in Section 4.2. Algorithm 9 shows a modified version which works with the modified split lists of Algorithm 8.

**Algorithm 9** Suffix rearrangements in the first phase of the algorithm. Variables $gs$ and $ge$ contain start and end bounds of the current group, $ps$ and $pe$ contain the start position of the first suffix set, directly after the execution of Algorithm 8.

```
1: for k = 1 up to l do                          ▷ iterate subsets P_1, ..., P_k in descending order
2:     ps ← pe
3:     pe ← pe + GSIZE[pe]                                              ▷ get bounds for set

4:     for i ← pe − 1 down to ps do   ▷ decrement group counts of previous suffixes
5:         p ← SA[i]
6:         sr ← GLINK[p]
7:         GSIZE[sr] ← GSIZE[sr] − 1
8:         sr ← sr + GSIZE[sr]
9:         s ← SA[sr]          ▷ move p to back of its group by exchange with last suffix
10:        pr ← ISA[p]
11:        SA[pr] ← s      ISA[s] ← pr
12:        SA[sr] ← p      ISA[p] ← sr
13:    end for

14:    for i ← pe − 1 down to ps do          ▷ set new GLINK for rearranged suffixes
15:        p ← SA[i]
16:        pgs ← GLINK[p]
17:        pgs ← pgs + GSIZE[pgs]
18:        GLINK[p] ← pgs
19:    end for

20:    for i ← pe − 1 down to ps do                        ▷ set GSIZE for new groups
21:        p ← SA[i]
22:        pgs ← GLINK[p]
23:        GSIZE[pgs] ← GSIZE[pgs] + 1
24:    end for
25: end for
26: SA[ge] ← gs                                              ▷ Preparation for Phase 2
```

The last step of the first phase is to prepare the current group for processing in the second phase, by setting $\mathsf{ISA}[s] \leftarrow ge$ for all suffixes $s$ of the group, and $\mathsf{SA}[ge] \leftarrow gs$. Both steps are performed within the Algorithms 8 and 9. The second phase can be implemented in exactly the same way as described in Section 4.2, Algorithm 6. The only thing to be considered is that prev pointers are contained in the GLINK array, not in a separate array.

Using these tricks, the algorithm can be implemented using $17n$ bytes of space: $n$ bytes for the text itself, $4n$ bytes for SA, and additional $12n$ bytes for ISA, GLINK and GSIZE, assuming that an integer requires 4 bytes. An implementation in C requires about 200 lines of code for the full algorithm, and can be found on github [2].

An additional space optimisation can be applied to the GSIZE array: GSIZE stores group sizes (and markings) whose distance is greater than or equal to their value. More detailed, let $\mathsf{GSIZE}[i] = k$ and $\mathsf{GSIZE}[j] = l$ with $i < j$ be any two numbers in GSIZE. Then the algorithm always ensures that $i+k \leq j$ holds. This permits the use of variable-length number storage in GSIZE, e.g. by using *Elias Gamma Coding* [5], and reduces the space requirements of GSIZE from $4n$ bytes to $2n$ bits.

Summing everything up, the algorithm can be implemented using $13.25n$ bytes of space. The implementation available on github [2] does not contain the variable-length number storage (basically because additional operations for fetching variable-length numbers would slow down the algorithm), thus it requires $17n$ bytes of space.

Beside space consumption, performance is a big point of interest on any algorithm. We already know that the proposed algorithm runs in asymptotic linear time, but theoretical runtime and practical performance can vary considerably. To fill this gap, the issue of the next chapter will be to measure practical performance, thus we will see whether the algorithm is suitable for real world usage.

# Chapter 6

# Performance Analyses

After the discussion about the algorithms functional principle, it is time to check if it is suitable for real world usage by comparing its performance with other suffix array construction algorithms.

A list of competing SACAs can be found in Table 6.1. It includes most common linear-time SACAs, such as the SA-IS by Nong et al., the algorithm by Ko and Aluru and the DC3 a.k.a. Skew Algorithm by Kärkkäinen and Sanders. Also, a very fast SACA named divsufsort is included, to compare results with the current state of the art.

Since the SACA described in this thesis makes heavy use of groups, and also works in a greedy manner, I called it GSACA—quite an imposing name for a suffix array construction algorithm, but results will tell whether the promising name satisfies its expectations.

| Algorithm | Resource | Runtime | Extra Working Space |
|---|---|---|---|
| divsufsort | [22, Yuta Mori] | $O(n \log n)$ | $O(1)$ |
| SAIS | [23, Yuta Mori] | $O(n)$ | $O(\log n) + \max 2n$ |
| KA | [15, Pang Ko] | $O(n)$ | $O(\log n) + 4.16n$ |
| DC3 | [29, Peter Sanders] | $O(n)$ | $O(\log n) + \max 24n$ |
| GSACA[1] | [2, This Thesis] | $O(n)$ | $O(1) + 12n$ |

Table 6.1: Used algorithms in benchmarks. Extra Working Space contains memory requirements for function calls ($O(1)$ for non–recursive, $O(\log n)$ for recursive algorithms), as well as space required in addition to the $5n$ for the text and the suffix array. All space requirements are measured in bytes, where an integer is assumed to require 4 bytes of space.

Now, a word to the test data. It is a selection of three text corpuses, namely the *Silesia Corpus* [4] with small–sized files ($< 40$ MB), the *Pizza & Chili Corpus* [7] with medium–sized files and the *Repetitive Corpus* [8] with highly repetitive files. The text selection contains texts with quite different types: normal texts, source codes, database data, html/xml documents as well as dna and protein sequences. This text variety gives a good overview over SACA performance in different contexts, and should lead to representative results in suffix array construction.

[1] $8.25n$ bytes of extra working space are possible using the variable-length storage described in Chapter 5, but additional operations would slow down the algorithm.

All experiments were conducted on a 64 bit Ubuntu 14.04.3 LTS (Kernel 3.13) system equipped with two ten-core Intel Xeon processors E5-2680v2 with 2.8 GHz and 128GB of RAM. The benchmark itself was compiled using g++ (version 4.8.4) and the -O3 option. Construction speeds[2] were measured using C++ built–in timers, while cache miss rates[3] were measured using `perf_events`[4]. Results (average of 10 runs) can be found in Tables 6.2, 6.3 and 6.4.

| File | Measurement | divsufsort | SAIS | KA | DC3 | GSACA |
|---|---|---|---|---|---|---|
| webster | speed[2] | 12.1 MB/s | 12.4 MB/s | 5.6 MB/s | 1.9 MB/s | 3.4 MB/s |
| $n =$ 39.5 MB, $\sigma =$ 98 | cache misses[3] | 39.6 % | 46.2 % | 32.6 % | 68.3 % | 68.2 % |
| dickens | speed[2] | 16.9 MB/s | 17.6 MB/s | 8.2 MB/s | 3.5 MB/s | 5.1 MB/s |
| $n =$ 9.7 MB, $\sigma =$ 100 | cache misses[3] | 6.5 % | 6.4 % | 7.4 % | 28.3 % | 49.8 % |
| nci | speed[2] | 18.7 MB/s | 21.5 MB/s | 10.6 MB/s | 3.3 MB/s | 5.0 MB/s |
| $n =$ 32.0 MB, $\sigma =$ 63 | cache misses[3] | 33.5 % | 45.5 % | 32.6 % | 59.6 % | 65.5 % |

Table 6.2: Benchmark results of the Silesia Corpus.

| File | Measurement | divsufsort | SAIS | KA | DC3 | GSACA |
|---|---|---|---|---|---|---|
| sources.200MB | speed[2] | 11.5 MB/s | 10.3 MB/s | 4.2 MB/s | 1.1 MB/s | 3.2 MB/s |
| $n =$ 200.0 MB, $\sigma =$ 231 | cache misses[3] | 55.4 % | 69.7 % | 52.6 % | 83.2 % | 74.0 % |
| dblp.xml.200MB | speed[2] | 10.9 MB/s | 10.3 MB/s | 4.1 MB/s | 1.3 MB/s | 3.3 MB/s |
| $n =$ 200.0 MB, $\sigma =$ 97 | cache misses[3] | 47.4 % | 76.0 % | 59.2 % | 90.0 % | 79.5 % |
| dna.200MB | speed[2] | 8.2 MB/s | 6.8 MB/s | 3.5 MB/s | 1.1 MB/s | 2.9 MB/s |
| $n =$ 200.0 MB, $\sigma =$ 17 | cache misses[3] | 42.9 % | 77.0 % | 54.8 % | 82.8 % | 82.0 % |
| proteins.200MB | speed[2] | 7.2 MB/s | 5.9 MB/s | 2.7 MB/s | 0.9 MB/s | 2.7 MB/s |
| $n =$ 200.0 MB, $\sigma =$ 26 | cache misses[3] | 47.4 % | 74.4 % | 55.0 % | 88.2 % | 80.3 % |
| english.200MB | speed[2] | 8.1 MB/s | 7.2 MB/s | 2.9 MB/s | 0.9 MB/s | 2.7 MB/s |
| $n =$ 200.0 MB, $\sigma =$ 226 | cache misses[3] | 54.4 % | 77.1 % | 54.2 % | 86.4 % | 79.2 % |

Table 6.3: Benchmark results of the Pizza & Chili Corpus.

| File | Measurement | divsufsort | SAIS | KA | DC3 | GSACA |
|---|---|---|---|---|---|---|
| world leaders | speed[2] | 24.1 MB/s | 22.3 MB/s | 9.7 MB/s | 2.8 MB/s | 4.3 MB/s |
| $n =$ 44.8 MB, $\sigma =$ 90 | cache misses[3] | 28.1 % | 37.6 % | 31.7 % | 59.0 % | 68.5 % |
| para | speed[2] | 8.4 MB/s | 11.0 MB/s | 3.1 MB/s | 1.2 MB/s | 3.0 MB/s |
| $n =$ 409.4 MB, $\sigma =$ 6 | cache misses[3] | 44.8 % | 83.5 % | 62.4 % | 88.1 % | 83.3 % |
| influenza | speed[2] | 10.5 MB/s | 13.6 MB/s | 5.4 MB/s | 1.7 MB/s | 3.7 MB/s |
| $n =$ 147.6 MB, $\sigma =$ 16 | cache misses[3] | 39.2 % | 78.5 % | 54.8 % | 83.0 % | 78.9 % |
| coreutils | speed[2] | 10.3 MB/s | 13.0 MB/s | 4.4 MB/s | 1.3 MB/s | 3.4 MB/s |
| $n =$ 195.8 MB, $\sigma =$ 237 | cache misses[3] | 55.1 % | 73.0 % | 51.7 % | 82.3 % | 72.6 % |
| Escherichia Coli | speed[2] | 9.4 MB/s | 11.1 MB/s | 4.1 MB/s | 1.4 MB/s | 3.0 MB/s |
| $n =$ 107.5 MB, $\sigma =$ 16 | cache misses[3] | 42.5 % | 70.5 % | 47.8 % | 77.8 % | 81.1 % |

Table 6.4: Benchmark results of the Repetitive Corpus.

[2]Construction speed: size of input/time to construct SA, in MB/s.
[3]Cache miss rate: number of cache misses/number of cache accesses of last–level cache, in %.
[4]`http://web.eece.maine.edu/~vweaver/projects/perf_events/`, last visited 10/2015.

The results clearly show that divsufsort and SAIS are playing in their own league: Construction speed is about 2 to 3 times faster than that of KA, DC3 and GSACA. Additionally, as Table 6.1 shows, the working space required by both of these algorithms is significantly smaller than that of KA, DC3 and GSACA.

Among the latter three algorithms, KA performs best, followed by GSACA. The DC3 algorithm performs worst of all algorithms, but this is owed to a very simple implementation requiring only about 50 lines of code. There might exist better implementations, but I wasn't able to find one running in linear time. Therefore, the bad results of DC3 shouldn't be overrated.

Let's have a closer look at the algorithm presented in this thesis, GSACA. To be honest, it performs quite poorly compared to the fastest algorithms presented here: it requires a lot of extra working space, and is about 2 to 5 times slower than SAIS or divsufsort. On the other hand, such results are not really surprising: the algorithm uses a lot of dependent non–parallelizable memory accesses, affecting quite different memory locations. As a direct conclusion, GSACA has the highest average cache miss rates of all algorithms except for the DC3 algorithm. Also note that even for very small–sized files (dickens from the Silesia Corpus), the cache miss rates produced by GSACA are orders of magnitude higher than that of other algorithms.

Cache miss rates of course are not the only reason for bad performance—as one can see, SAIS performs well despite high cache miss rates on bigger files—but high rates in consumption with a big working–constant lead to long construction times. So truly, compared to the state of the art, GSACA will not be used in real world applications at the current stage—the competitors clearly outperform GSACA.

# Chapter 7

# Conclusion

This thesis has presented a new approach for linear-time suffix array construction. A new suffix sorting principle was introduced, leading to the first non-recursive linear-time suffix array construction algorithm, GSACA.

Unfortunately, GSACA cannot hold up with current state of the art suffix array construction algorithms: Construction speed is about 2 to 5 times slower than that of faster algorithms, and the extra working space consumption is quite large.

As a result, the algorithm presented in this thesis is interesting only from a theoretical point of view, since it is the first non-recursive linear-time SACA. Nonetheless, the new sorting principle may be used to design better linear-time algorithms. To give some ideas at least, GSACA deals a lot with previous smaller and next smaller values, what hints to a stack-based computation for next lexicographically smaller suffixes. The group belonging of a suffix may be computed 'on the fly', during the computation of next lexicographically smaller suffixes. Finally, another representation of suffix groups may leads to a cache-friendly implementation of the second phase, resulting in a much faster algorithm. However, these ideas are suggestions, no specific description of a better algorithm.

Summarizing, the results of this thesis are quite promising: Compared to the developmental history of the currently fastest linear-time suffix array construction algorithm SA-IS, development of GSACA is in its infancy. Thus, there's a lot of room for improvements, so it's worth using GSACA as a starting point for future research topics.

# Bibliography


[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing Suffix Trees with Enhanced Suffix Arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] U. Baier. GSACA Algorithm. `https://github.com/waYne1337/gsaca`. last visited 10/2015.

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

[4] S. Deorowicz. Silesia Corpus. `http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia`. last visited 10/2015.

[5] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[6] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Proceedings of the 41th Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 390–398, 2000.

[7] P. Ferragina and G. Navarro. Pizza & Chili Corpus. `http://pizzachili.dcc.uchile.cl/texts.html`. last visited 10/2015.

[8] P. Ferragina and G. Navarro. Repetitive Corpus. `http://pizzachili.dcc.uchile.cl/repcorpus.html`. last visited 10/2015.

[9] J.-L. Gailly and M. Adler. `http://www.gzip.org/`. last visited 7/2015.

[10] R. Grossi and G. F. Italiano. Suffix trees and their applications in string algorithms. In *Proceedings of the 1st South American Workshop on String Processing*, pages 57–76, 1993.

[11] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '03, pages 251–260, 2003.

[12] J. Kärkkäinen and P. Sanders. Simple Linear Work Suffix Array Construction. In *Proceedings of the 30th International Conference on Automata, Languages and Programming*, ICALP '03, pages 943–955, 2003.

[13] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear Work Suffix Array Construction. *Journal of the ACM*, 53(6):918–936, 2006.

[14] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time Construction of Suffix Arrays. In *Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching*, CPM '03, pages 186–199, 2003.

[15] P. Ko. Ko–Aluru Algorithm. `https://sites.google.com/site/yuta256/KA.tar.bz2`. last visited 10/2015.

[16] P. Ko and S. Aluru. Space Efficient Linear Time Construction of Suffix Arrays. In *Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching*, CPM '03, pages 200–210, 2003.

[17] S. Kreft and G. Navarro. On Compressing and Indexing Repetitive Sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[18] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching*, CPM '13, pages 189–200, 2013.

[19] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Parallel External Memory Suffix Sorting. In *Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching*, CPM '15, pages 329–342, 2015.

[20] U. Manber and G. Myers. Suffix Arrays: A New Method for On-line String Searches. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 319–327, 1990.

[21] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[22] Y. Mori. libdivsufsort. `https://github.com/y-256/libdivsufsort`. last visited 10/2015.

[23] Y. Mori. sais–lite–2.4.1. `https://sites.google.com/site/yuta256/sais`. last visited 10/2015.

[24] Y. Mori. Suffix Array Construction Benchmark. `https://code.google.com/p/libdivsufsort/wiki/SACA_Benchmarks`, 2008. last visited 7/2015.

[25] J. C. Na. Linear-Time Construction of Compressed Suffix Arrays Using O(N Log N)-bit Working Space for Large Alphabets. In *Proceedings of the 16th Annual Conference on Combinatorial Pattern Matching*, CPM '05, pages 57–67, 2005.

[26] G. Nong, S. Zhang, and W. H. Chan. Linear Suffix Array Construction by Almost Pure Induced-Sorting. In *Proceedings of the 2009 Data Compression Conference*, DCC '09, pages 193–202, 2009.

[27] G. Nong, S. Zhang, and W. H. Chan. Two Efficient Algorithms for Linear Time Suffix Array Construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011.

[28] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A Taxonomy of Suffix Array Construction Algorithms. *ACM Computational Survey*, 39(2), 2007.

[29] P. Sanders. DC3 Algorithm. `http://people.mpi-inf.mpg.de/~sanders/programs/suffix/`. last visited 10/2015.

[30] J. Seward. `http://www.bzip.org/`. last visited 7/2015.

[31] P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, SWAT '73, pages 1–11, 1973.

[32] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

Name: Uwe Baier Matriculation Number: 721798

**Declaration**

I declare that I have developed and written the enclosed Master Thesis completely by myself, and have not used sources or means without declaration in the text. Any thoughts from others or literal quotations are clearly marked. The Master Thesis was not used in the same or in a similar version to achieve an academic grading or is being published elsewhere.

Ulm, the .....

Uwe Baier