WORST-CASE LINEAR TIME SUFFIX ARRAY CONSTRUCTION ALGORITHMS

MIKKEL RAVNHOLT KNUDSEN, 201303546
JENS CHRISTIAN LIINGAARD HANSEN, 201303561

MASTER’S THESIS
June 2018
Advisor: Christian Nørgaard Storm Pedersen
WORST-CASE LINEAR TIME SUFFIX ARRAY CONSTRUCTION ALGORITHMS

MIKKEL RAVNHOLT KNUDSEN, JENS CHRISTIAN LIINGAARD HANSEN

Master’s Thesis
Department of Computer Science
Science & Technology
Aarhus University

June 2018
This thesis intends to explore different algorithms for constructing suffix arrays, focusing in particular on worst-case linear time algorithms. These algorithms are described in detail, including their similarities and differences, using one common notation. These descriptions clarify and elaborate on parts, including correctness proofs, of the original papers that were deemed inadequately detailed.

The algorithms being explored are a naive way to construct suffix arrays, the KA algorithm by Ko and Aluru [1], the Skew algorithm by Kärkkäinen and Sanders [2], the SA-IS algorithm by Nong, Zhang, and Chan [3] and the MP algorithm by Maniscalco and Puglisi [4]. The KA, Skew and SA-IS algorithms have been chosen because they are all worst-case linear time algorithms, each contributing new ideas to the field of suffix array construction. To contrast these algorithms, we have included the MP algorithm, which focuses on being fast in practice, rather than having an optimal worst-case time complexity.

These algorithms have been implemented based on our interpretation of the descriptions from their associated papers, and any issues and intricacies encountered have been explained thoroughly.

To facilitate the testing of these algorithms, various types of artificial data has been created, in an attempt to show the best- and worst-case input for each algorithm, and to investigate how the algorithms compare against one another. Furthermore, all algorithms have been tested against a collection of real world DNA sequences that have been widely used to test the performance of suffix array construction algorithms in the past.

The experiments conducted serve as a frame of reference for comparison of the chosen suffix array construction algorithms, and the hope is that the results generalize to data not explored in this thesis.

We conclude that for sufficiently large input, it is generally the case that the MP algorithm outperforms the SA-IS algorithm, which outperforms the KA algorithm, which outperforms the Skew algorithm. The MP algorithm outperforms all other algorithms for sufficiently large real world data, suggesting that algorithms with super-linear worst-case running times, engineered to work well in practice, might very well be faster than asymptotically optimal algorithms in real world applications.
## CONTENTS

I  INTRODUCTION AND PRELIMINARIES  1

1  INTRODUCTION  3
   1.1  Motivation ...........................................  3
   1.2  Focus and contribution ..............................  3

2  PRELIMINARIES  7
   2.1  Alphabets ...........................................  7
   2.2  Strings ...............................................  7
   2.3  Order of strings ...................................  8
   2.4  Suffix array .........................................  9

II  ALGORITHMS AND IMPLEMENTATIONS  11

3  GENERAL IMPLEMENTATION REMARKS  13
   3.1  Exceeding maximum recursion depth .................  13
   3.2  Implementation consistency ........................  13
   3.3  Correctness tests .................................  13

4  THE NAIVE ALGORITHM  15
   4.1  Algorithm description .............................  15
   4.2  Time complexity ...................................  15
   4.3  Implementation notes ..............................  16

5  SACAS BASED ON INDUCED SORTING  17

6  THE KA ALGORITHM  19
   6.1  Notation and preliminaries ........................  19
   6.2  Algorithm description .............................  21
   6.3  Time complexity ...................................  27
   6.4  Correctness .........................................  29
   6.5  Implementation notes ..............................  31

7  THE SKEW ALGORITHM  35
   7.1  Algorithm description .............................  35
   7.2  Time complexity ...................................  37
   7.3  Correctness .........................................  38
   7.4  Implementation notes ..............................  40

8  THE SA-IS ALGORITHM  43
   8.1  Preliminaries .......................................  43
   8.2  Algorithm description .............................  44
   8.3  Time complexity ...................................  46
   8.4  Correctness .........................................  47
   8.5  Implementation notes ..............................  48

9  THE MP ALGORITHM  49
   9.1  Algorithm description .............................  49
   9.2  Time complexity ...................................  51
   9.3  Correctness .........................................  53
9.4 Further improvements .......................... 54
9.5 Implementation notes ......................... 55

III EXPERIMENTAL EVALUATION

10 TESTING METHODOLOGY
  10.1 Warm up ..................................... 59
  10.2 Garbage collection ............................ 59
  10.3 Timing and plotting ........................... 59

11 ARTIFICIAL DATA
  11.1 About the data ............................... 61
  11.2 Experiments on individual algorithms ........ 62
  11.3 Experiments comparing all algorithms ......... 72

12 REAL WORLD DATA

13 EXPERIMENT EVALUATION AND DISCUSSION

14 CONCLUSION AND FUTURE WORK
  14.1 Conclusion .................................. 85
  14.2 Future work ................................. 86

BIBLIOGRAPHY .................................. 87
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Suffix array with corresponding buckets</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>String with S/L-types assigned</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>$S_S$ and $S_L$</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>Placement of S- and L-buckets</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>S-substrings illustrated with intervals. Note that neighbouring S-substrings share endpoints.</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>String with corresponding S-distances</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>Steps of the proof summarized</td>
<td>31</td>
</tr>
<tr>
<td>8</td>
<td>Example of suffixes from $S^2$ with their corresponding 3-grams and ranks</td>
<td>36</td>
</tr>
<tr>
<td>9</td>
<td>Figure 8 extended with $A$, $B$, $S'$ and $\tau(i)$</td>
<td>40</td>
</tr>
<tr>
<td>10</td>
<td>Example of LMS-characters, -substrings and -suffixes</td>
<td>44</td>
</tr>
<tr>
<td>11</td>
<td>Running time of naive algorithm on different inputs. Note that MaximumLMS, MaximumL, MaximumS, Hetero and Random lie almost on top of one another</td>
<td>63</td>
</tr>
<tr>
<td>12</td>
<td>Running time of naive algorithm on different inputs, divided by $n^2 \cdot \log n$</td>
<td>63</td>
</tr>
<tr>
<td>13</td>
<td>Running time of KA algorithm on different inputs. Note that MaximumL and MaximumS practically lie on top of one another</td>
<td>65</td>
</tr>
<tr>
<td>14</td>
<td>Running time of KA algorithm on different inputs, divided by $n$</td>
<td>65</td>
</tr>
<tr>
<td>15</td>
<td>Running time of Skew algorithm on different inputs. Note that MaximumL and MaximumS practically lie on top of one another</td>
<td>66</td>
</tr>
<tr>
<td>16</td>
<td>Running time of Skew algorithm on different inputs, divided by $n$</td>
<td>67</td>
</tr>
<tr>
<td>17</td>
<td>Running time of SA-IS algorithm on different inputs. Note that MaximumL, MaximumS and Repetitive20 practically lie on top of one another</td>
<td>68</td>
</tr>
<tr>
<td>18</td>
<td>Running time of SA-IS algorithm on different inputs, divided by $n$</td>
<td>68</td>
</tr>
<tr>
<td>19</td>
<td>Running time of MP algorithm on different input. Note that MaximumLMS and Random practically lie on top of one another, as does MaximumL and MaximumS</td>
<td>70</td>
</tr>
<tr>
<td>20</td>
<td>Running time of MP algorithm on different types of input, divided by $n$</td>
<td>70</td>
</tr>
</tbody>
</table>
Figure 21  Running time of all algorithms on heterogeneous input  .......... 72
Figure 22  Running time of all algorithms on homogeneous input .......... 73
Figure 23  Running time of all algorithms on MaximumL input .......... 74
Figure 24  Running time of all algorithms on MaximumS input .......... 75
Figure 25  Running time of all algorithms on Maximum-LMS input. Note that Skew and SA-IS practically lie on top of one another .......... 76
Figure 26  Running time of all algorithms on random input .......... 77
Figure 27  Running time of KA, SA-IS and Skew on repetitive input with a repetition length of 100 ........ 78
Part I

INTRODUCTION AND PRELIMINARIES

In this part we introduce the topic of the thesis as well as our focus and contribution. Furthermore, we go through the preliminaries necessary for the following parts of the thesis.
INTRODUCTION

The overall topic of this thesis is suffix array construction algorithms, or SACAs for short, with particular attention paid to worst-case linear time SACAs.

In this chapter we motivate the subject of suffix arrays and their construction, introduce the algorithms that will be treated in this thesis and clarify what our focus and contribution is.

1.1 MOTIVATION

The first question one might ask is why suffix arrays are even interesting in the first place, so we start by briefly motivating suffix arrays and consequently efficient algorithms for constructing them.

The suffix array is a data structure closely related to the suffix tree, which is a versatile and widely used data structure in various string processing applications, in particular text indexing. However, a major drawback of the suffix tree is its space usage, both during construction and afterwards in storage, which makes it impractical for large strings. This limitation was one of the primary reasons the suffix array was introduced as an alternative to the suffix tree in 1993 by Manber and Myers [5]. Even efficient suffix tree implementations use significantly more space than their corresponding suffix array [6].

In a paper by Abouelhoda, Kurtz, and Ohlebusch [7] from 2004, it was shown that the suffix array, possibly enhanced with some auxiliary data, can in fact do everything the suffix tree can with the same time complexity. This, coupled with the space advantage, means that a substantial amount of research has gone into algorithms that construct suffix arrays. A suffix array can be created in linear time from the corresponding suffix tree via a DFS traversal and since the suffix tree of a given string can be constructed in linear time, (see e.g. [8] and [9]) so can the suffix array. However, then the space advantage during construction would be lost so researchers have mainly focused on algorithms that construct the suffix array directly, that is, algorithms that are both time and space efficient.

1.2 FOCUS AND CONTRIBUTION

The literature in the field of suffix array construction algorithms is quite substantial and so the purpose of this section is to clarify what we focus on in this thesis and, just as importantly, what we do not focus on.
In 2006 a SACA overview paper was published by Puglisi, Smyth, and Turpin [10]. This paper describes a number of SACAs that had been published up until that point. It also gives an overview of the real world performance of these algorithms, based on a mixture of implementations by the authors and implementations obtained elsewhere. Because the purpose of the paper was to provide a broad overview, it does not go much further than describing the broad ideas of the algorithms it presents. In particular, detailed arguments of correctness as well as remarks on the feasibility of implementing the algorithms are not given.

After reading this paper one gets the impression that SACA research can be roughly divided into two main categories:

- The development of algorithms that are asymptotically optimal, that is algorithms that run in $\Theta(n)$ time in the worst case*.

- The development of algorithms that are fast in practice, that is algorithms that on real-world input are fast in absolute terms. These algorithms are not necessarily asymptotically optimal.

In this thesis we have selected three worst-case linear time SACAs as our main focus, two of which occur in the overview paper. Here we briefly introduce these algorithms and their distinguishing features:

- The KA algorithm by Ko and Aluru [1]: This algorithm was among the first worst-case linear SACAs. It introduced the idea of splitting suffixes into one of two types, such that the sorting of one type can be used to induce the sorting of the other. We treat this algorithm in Chapter 6.

- The Skew algorithm by Kärkkäinen and Sanders [2]: This worst-case linear algorithm appeared about the same time as the KA algorithm and also works by splitting suffixes into different types. However, unlike the KA algorithm, these types are not based on the contents of the suffixes, but rather on their indices. We return to this algorithm in Chapter 7.

- The SA-IS algorithm by Nong, Zhang, and Chan [3]: This algorithm, which also runs in linear time in the worst case, is the most recent of the algorithms we consider. It builds on top of ideas introduced by KA, but provides new insights and improvements, yielding a simpler and arguably more elegant algorithm. The SA-IS algorithm is treated in Chapter 8.

The holy grail, so to speak, of SACAs is an algorithm that is both asymptotically optimal, like the three algorithms above, and fast in

*Any SACA must always at least read its entire input, as well as write to every entry of the suffix array, meaning it has to spend $\Omega(n)$ time on any input. That is why $\Theta(n)$ is the best time complexity any SACA can hope to achieve.
practice, i.e. on data encountered in the real world. In the previously mentioned overview article by Puglisi, Smyth, and Turpin [10], the authors found that the fastest algorithm in practice was the MP algorithm by Maniscalco and Puglisi [4]. This algorithm differs from the previous three algorithms by treating suffixes as independent strings, which it then sorts using specialized string sorting procedures and heuristics in order to achieve fast running times.

We have included this algorithm in our selection of algorithms because it represents the other category of SACAs that are not asymptotically optimal and as such it is interesting to compare it to the three worst-case linear algorithms introduced above. Further, we want to investigate whether it is also fastest among our implementations, especially since the overview paper predates the SA-IS algorithm. We return to the MP algorithm in Chapter 9.

While these algorithms are very different in their details, they share a common overall structure which is described in Chapter 5. Finally, as a baseline, we also include a naive algorithm that uses Java’s built-in array sorting procedure. This is described in Chapter 4.

With this total of five algorithms as a starting point, our contribution is two-fold:

1. We describe the selected algorithms in detail using a common language and terminology. In these descriptions we highlight the similarities and differences between the algorithms and also provide extra detail or clarification in parts we personally found to be insufficiently detailed in the original papers, including correctness proofs. The hope is that this will give a more unified picture of these algorithms and ease the understanding process for people who wish to study these algorithms in the future.

2. We implement each of the selected algorithms in Java and describe any significant issues we encountered while translating the algorithm descriptions into working implementations. In Part iii we evaluate our implementations experimentally on both artificial and real-world data, compare their relative performance and discuss the results. For the algorithms that also appear in the overview paper [10], we also compare our findings to theirs.

Although space usage, that is, the actual number of bytes used per input character, is a main motivation for using suffix arrays, we will restrict our focus to the time usage of the selected algorithms in both the absolute and asymptotic sense. The reason for this is that while all the algorithms use a linear amount of space, reducing the actual number of bytes per input character comes down to meticulously optimizing the algorithms at the implementation level. While this becomes important in practice for very large strings, we will not pay any further attention to it, since such optimizations are largely
independent of the individual algorithms and thus do not contribute much towards understanding how and why these work.
Before we proceed, we first introduce some basic definitions and notation related to strings and suffix arrays that we will use throughout the thesis.

### 2.1 Alphabet

**Definition 2.1 (Alphabet)**

An alphabet, $\Sigma$, is a totally ordered set of characters*, i.e. for any two distinct symbols $a, b \in \Sigma$ it is either the case that $a < b$ or $b < a$. Furthermore, we assume that this relative order can be determined in constant time.

In this thesis we assume that given a string $S \in \Sigma^*$ of length $n$, the alphabet satisfies that $|\Sigma| \in O(n)$. We will refer to such an alphabet as a linear alphabet.

When implementing string algorithms, it can be convenient to think of a given alphabet as consisting of the integers $\{0, 1, \ldots, |\Sigma| - 1\}$, where $0$ replaces the smallest character of $\Sigma$, $1$ the second smallest etc. Any alphabet can be converted to such an equivalent integer alphabet by iterating through $\Sigma$ in order.

### 2.2 Strings

**Definition 2.2 (String)**

Given an alphabet, $\Sigma$, a string, $S$, is a finite sequence of $n$ symbols from $\Sigma$. $\Sigma^*$ is defined to be the infinite set of all possible strings over $\Sigma$.

Given a string $S$, we take $S[i]$ to mean the character at position $i$ in $S$ where the first character has index 0.

#### 2.2.1 Substrings

**Definition 2.3 (Substring)**

Given a string $S$, we define $S[i \ldots j]$, where $i \leq j$, to mean the string $S[i] \circ S[i+1] \circ \ldots \circ S[j]$, where $\circ$ denotes concatenation. Since this string occurs in $S$ we call it a substring of $S$.

---

*The terms ‘character’ and ‘symbol’ will be used interchangeably when referring to an element of an alphabet*
2.2.2 Prefixes and suffixes

Given a string $S \in \Sigma^*$ of length $n$, prefixes of $S$ are defined as follows:

**Definition 2.4 (String prefix)**
The substring $S[0 \ldots i]$, for $0 \leq i < n$, is called the $i$th prefix of $S$. If $i < n - 1$, it is called a proper prefix.

Similarly, one can define suffixes:

**Definition 2.5 (String suffix)**
The substring $S[i \ldots n - 1]$, for $0 \leq i < n$, is called the $i$th suffix of $S$ and we use the shorthand notation $S_i$ to refer to this suffix.

2.2.3 Sentinel

In the rest of this thesis we will assume that a given string, $S$, is always terminated by a special character, denoted with $\$, often called a sentinel. It is defined as follows:

**Definition 2.6 (Sentinel)**
The sentinel is a character that only occurs as the last character in a given string $S$ and is smaller than every other character in $S$.

The assumption that all strings are terminated by a sentinel is convenient because it eliminates certain special cases, e.g. that one suffix can be a prefix of a different suffix. The assumption is, however, not strictly necessary in the sense that the algorithms we will present could be adapted to work without it, at the cost of being more complex and less readable.

2.3 ORDER OF STRINGS

When we have an ordered alphabet, we can use this order to define an order on $\Sigma^*$.

**Definition 2.7 (Lexicographical string order)**
Strings are ordered lexicographically. That is, for two different strings $U, V \in \Sigma^*$, we say that $U$ is smaller than $V$ if and only if one of the two following conditions is satisfied:

1. $U$ is a proper prefix of $V$

2. There exists an index $i$ such that $U[i] < V[i]$ and $U[j] = V[j]$ for every $0 \leq j < i$

If neither $U < V$ nor $V < U$ holds, then $U = V$.

When we talk about sorting strings, it will always mean sorting in ascending order.
2.4 SUFFIX ARRAY

Definition 2.8 (Suffix array)
Given a string $S$ of length $n$, the suffix array $SA$ of $S$ is a permutation of the integers from 0 to $n - 1$ satisfying that $S_{SA[i]} < S_{SA[i+1]}$ for all $i$ satisfying $0 \leq i < n - 1$.

That is, the suffix array of $S$ provides the sorted order of all suffixes of $S$. Note that because all suffixes of $S$ differ in length, no two suffixes are equal, and so, there can only be one correct suffix array of $S$.

Definition 2.9 (Inverse suffix array)
The inverse suffix array of $S$, referred to as $ISA$, is a permutation of size $n$, satisfying the equality $ISA[SA[k]] = k$ for every $k$, $0 \leq k < n$.

That is, $ISA[i] = j$ means that $S_i$ is the $j^{th}$ smallest suffix of $S$. Note that $ISA$ can be made from $SA$ in linear time, by scanning $SA$ once.

2.4.1 Suffix array buckets

Because the suffix array, $SA$, of a string, $S$, provides the lexicographical ordering of the suffixes of $S$, it is clear that all suffixes starting with the same character will appear in a contiguous interval of $SA$. That is, if our alphabet is the set $\{a, b, c\}$, all suffixes starting with $a$ will appear before all suffixes starting with $b$ which in turn will appear before all suffixes starting with $c$.

This means that for every character $c \in S$ we can define what we will refer to as the bucket of $c$:

Definition 2.10 (Suffix array bucket)
For a character $c \in S$ we define the bucket of $c$ be the contiguous entries of $SA$ in which all suffixes starting with $c$ occur.

For instance, in Figure 1, all suffixes starting with $c$ will be placed in the third bucket from the left.

\[
\text{SA: } \begin{array}{ccccccc}
\_ & \_ & \_ & \_ & \_ & \_ & \_ \\
\text{a-bucket} & \text{b-bucket} & \text{c-bucket} & \_ & \_ & \_ & \_ \\
\end{array}
\]

Figure 1: Suffix array with corresponding buckets
Part II

ALGORITHMS AND IMPLEMENTATIONS

In this part we describe and compare five different suffix array construction algorithms. In addition to describing how and arguing why each of them work, we also focus on their similarities and differences.

The descriptions of the algorithms broadly follow those found in the original papers, but have been changed to use the same common notation throughout this thesis. Furthermore, any gaps and complicated details have been explained in greater detail. Since the algorithms build on similar ideas, we have made a number of correctness proofs ourselves and rewritten a number of existing proofs to reflect our notation and fill out any gaps we encountered.

Finally, we describe any issues encountered while translating the associated papers into working implementations and how we handled these issues.
The purpose of this chapter is to present and discuss some general implementation notes that are not related to one particular algorithm. Our implementations are available at [https://github.com/j46/speciale](https://github.com/j46/speciale).

### 3.1 Exceeding Maximum Recursion Depth

In Java, as in many other programming languages, a function call involves the allocation of a new stack frame (see [11], section 2.6). This in turn means that whenever one writes a recursive function, there is a risk of a stack overflow if the recursion depth becomes too great.

Three of the four recursive algorithms we have implemented have logarithmic recursion depth\(^*\) and so there is no risk of a stack overflow for any realistically sized inputs. The last of these four recursive algorithms is the MP algorithm which, in its most basic form, can get linear recursion depth in the worst case and so there is a risk of encountering a stack overflow error for particular inputs. However, this is handled explicitly and described in greater detail in Chapter 9.

### 3.2 Implementation Consistency

In order to keep our implementations consistent, as well as to avoid code duplication, we have made an effort to reuse code whenever possible. As an example, many of the algorithms calculate the start- and endpoints of the suffix array buckets and so we have encapsulated this computation in a single, shared function.

### 3.3 Correctness Tests

All algorithm implementations have been tested against one another, to see that, given some input, they all produce the same suffix array, as would be expected. We have used several input types in our correctness tests and we describe these input types further in Chapter 11, where we conduct experiments on our implementations using artificial data.

Since the naive algorithm is very simple to implement, we are confident that it produces the correct suffix array for a given input string. This, and the fact that all implemented algorithms produce the same

\(^*\)Specifically base 2 or 3/2 in the worst case.
result on all tested input, makes us confident that all algorithms produce correct results.
THE NAIVE ALGORITHM

4.1 ALGORITHM DESCRIPTION

Suffix arrays can be constructed in a straightforward manner, by simply creating an array of all suffixes and sorting each suffix, using some comparison-based algorithm where two suffixes are compared character-wise, from left to right, in order to determine their lexicographical order. This naive algorithm can be easily improved by comparing suffixes of a given string without explicitly creating a new substring for each suffix, thus getting rid of the $\Theta(n^2)$ time and space usage for string allocation. We will henceforth refer to this algorithm simply as the naive algorithm.

4.2 TIME COMPLEXITY

Comparison-based sorting requires $\Omega(n \log n)$ suffix comparisons in the worst case ([12], p. 193). Consider a sorting algorithm where this bound is tight, i.e. an algorithm that uses $\Theta(n \log n)$ comparisons in the worst case.

Unlike when sorting e.g. integers, where a comparison can be performed in $O(1)$ time*, the comparison of two suffixes can take anywhere from constant to linear time.

The best-case input string is a string where each character occurs only once, i.e. a string of the form $S = a_0 \circ a_1 \circ \ldots \circ a_{n-1}$ where $a_i \neq a_j$ for $i, j \in \{0, \ldots, n-1\}$, when $i \neq j$. In this case, each suffix comparison only takes $O(1)$ time, since the ordering of two suffixes is always determined by their first character. For such an input string the worst-case running time of the sorting algorithm is $\Theta(n \log n) \cdot O(1) = \Theta(n \log n)$.

On the other hand, consider a string consisting of the same character repeated, i.e. $S = a \circ a \circ \ldots \circ a$ for some $a \in \Sigma$. Comparing two suffixes of such a string takes time proportional to the shortest of the two suffixes, since the order is not determined until the sentinel is reached. That is, a suffix comparison can take $\Theta(n)$ time for certain suffixes, for instance the first two. As such, $O(n^2 \log n)$ is an upper bound for the worst-case running time of the sorting algorithm.

*Assuming the integer fits in a (constant number of) machine word(s).
4.3 IMPLEMENTATION NOTES

We implemented the algorithm by using Java’s built-in array sorting algorithm [13], providing a custom comparison function. Given an input string $S$ and two indices $i, j$, this comparison function determines the order of $S_i$ and $S_j$ by scanning the two suffixes from left to right until their characters differ, which they always will eventually, because of the sentinel.
Over the years, several categories of suffix array construction algorithms have emerged \cite{10}, some of them being SACAs based on so-called \textit{induced sorting}. Algorithms in this category consist of the same overall steps, but often differ significantly in their details and techniques. These overall steps are:

1. \textit{Sample selection}: Select a special sample of the suffixes of the input string.

2. \textit{Sample sorting}: Sort this suffix sample.

3. \textit{Induced sorting}: Use the sorting of the suffix sample to induce the order of the remaining suffixes.

The remaining four algorithms we consider follow this structure, but differ in particular in how the sample suffixes are sorted. All the worst-case linear time algorithms we consider do this by creating a new, smaller string from the selected sample after which the suffix array for this new string is constructed recursively, if necessary. The suffix array of this smaller string is then used to sort the selected sample. The MP algorithm, however, does not sort the suffix sample in this manner.

Of course, in order for an algorithm based on induced sorting to be useful, these steps all have to be efficient either from a theoretical or practical point of view or, ideally, both. The algorithms we will look at use various techniques to achieve this goal.
The KA algorithm, made by Ko and Aluru [1] in 2003, is a worst-case linear time algorithm and was, along with the Skew algorithm which we treat in Chapter 7, among the first SACAs to achieve linear time complexity in the worst case. Given a string S, it works by first categorizing each suffix of S as either S-type or L-type as defined in Definition 6.1. It then takes all suffixes of the type of which there are fewest and sorts these. From the sorted order of these sample suffixes, the suffixes of the type of which there are most can be induced.

The description of the algorithm found in the original paper is quite vague and lacks detail in certain important parts, which meant that we had to fill in a significant amount of gaps when understanding and, in particular, when implementing the algorithm. Our first implementation was very clumsy, but during this work we had several realizations that made our understanding of the algorithm clearer. This allowed us to rework the implementation and make it significantly simpler and more succinct.

When we describe the algorithm in Section 6.2, we include details and realizations that, in our opinion, were not sufficiently described in the original paper but that we deem important. This includes stating certain things explicitly that are implicit in the original paper.

We return to the issues we faced during implementation in Section 6.5.

6.1 Notation and Preliminaries

**Definition 6.1 (S- and L-type suffixes and characters)**

Given a string S, define each suffix $S_i$ of S to be an S-type suffix, if it is the case that $S_i$ is lexicographically smaller than $S_{i+1}$ and an L-type suffix otherwise – i.e. S for smaller and L for larger. The last suffix, $S_{n-1}$ is a special case, since it has no suffix following it, and its type is defined differently, depending on the algorithm. The type of the last suffix will be stated explicitly, when necessary.

Furthermore, we call $S[i]$ an S-type character if $S_i$ is an S-type suffix – L-type characters are defined similarly.

Since all suffixes of a given string have different lengths, it will always be the case that $S_i < S_{i+1}$ or $S_i > S_{i+1}$, except for the last suffix, $S_{n-1}$. This means that, given the type of the last suffix, all other suffixes can be assigned to be either S- or L-type.

From the above, it can be seen that $S_i$ is an S-type suffix, if it is the case that $S[i] < S[i+1]$, or if $S[i] = S[i+1]$ and $S[i+1]$ is an S-
type suffix. Similarly, it can be seen that $S_i$ is an L-type suffix, if it is the case that $S[i] > S[i+1]$, or if $S[i] = S[i+1]$ and $S[i+1]$ is an L-type suffix. Using this, the type of all suffixes of a string $S$ can be computed in linear time as follows:

First, define the type of the last suffix. Then, scan $S$ from right to left, and for each character, check which of the two conditions is fulfilled.

An example of $S/L$-type assignment can be seen in Figure 2, where the last suffix has been defined to be S-type.

\begin{figure}[h]
\centering
\begin{tabular}{c|c}
\hline
Pos: & 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 \\
S/L: & L S L L S L L S L S S L L L S S $ \\
\hline
\end{tabular}
\caption{String with $S/L$-types assigned}
\end{figure}

Lemma 6.2 (L-type suffixes before S-type suffixes)

Given a string $S$, consider two suffixes $S_L$ and $S_S$ starting with the same character $c$. Further, let $S_L$ be an L-type suffix and $S_S$ be an S-type suffix. Then $S_L < S_S$.

Proof. Since both $S_S$ and $S_L$ start with $c$, we can write them on the form $S_S = c\alpha$ and $S_L = c\beta$ where $\alpha, \beta \in \Sigma^*$.

Per the definition of L-type suffixes, the fact that $S_L$ is L-type means that $c\beta > \beta$. Due to the sentinel, $\beta$ cannot be a prefix of $c\beta$ and so, per the definition of lexicographical ordering, there must exist some index $i \geq 0$ such that $c\beta[i] > \beta[i]$ and $c\beta[l] = \beta[l]$ for $0 \leq l < i$. What this means is that $\beta$ has the form $c^ib\beta'$ where $b$ is some character less than $c$.

By a similar argument, $\alpha$ has the form $c^jd\alpha'$ for some integer $j \geq 0$, and where $d$ is a character greater than $c$. See Figure 3 for an example of what $S_S$ and $S_L$ looks like.

\begin{figure}[h]
\centering
\begin{tabular}{c|c}
\hline
$S_S$ & : $c\alpha = ccc\ldots cd\ldots$ \\
$S_L$ & : $c\beta = ccc\ldots cb\ldots$ \\
\hline
\end{tabular}
\caption{$S_S$ and $S_L$}
\end{figure}

The order of $S_S$ and $S_L$ is determined by the order of $\alpha$ and $\beta$ because both suffixes start with the character $c$. Since $\alpha$ and $\beta$ are both suffixes ending with $\$$, there must exist some smallest index $k$, such that $\alpha[k] \neq \beta[k]$. Due to the form of $\alpha$ and $\beta$, $k$ must be either $i+1$ or $j+1$. Assume without loss of generality that $k = i+1$, meaning $i+1 \leq j+1$.

Because $k = i+1 \leq j+1$ we have that $\alpha[k] = d$, and $\beta[k] = c$ or $\beta[k] = b^*$. In either case $\alpha[k] > \beta[k]$ which means $\alpha > \beta$. Finally, this gives us that $S_S = c\alpha > c\beta = S_L$.

*If $\beta[k] = b$ if $k = j+1$ and $\beta[k] = c$ if $k < j+1$.

\end{document}
Because of this lemma, we can define S- and L-buckets:

**Definition 6.3 (S- and L-buckets)**

Given a string S and its suffix array SA we can, for a character \( c \in S \), subdivide its suffix array bucket into two contiguous parts. The first part contains all the L-type suffixes that start with \( c \), while the second part contains all the S-type suffixes starting with \( c \). We call the first part the L-type bucket and the second part the S-type bucket. This is exemplified in Figure 4.

![Figure 4: Placement of S- and L-buckets](image)

### 6.2 Algorithm Description

#### 6.2.1 Sample selection

For sample selection, the KA algorithm first determines the S/L-type of each suffix in S except the last one, \( S_{n-1} \), as described above. As the suffix sample it then selects all suffixes of the type of which there are fewest. That is, if there are fewer S-type suffixes than L-type, it chooses all S-type suffixes for sample sorting.

In the rest of this section assume that the sample suffixes are all the S-type suffixes. In Section 6.2.4 we return to the case where the sample suffixes are of type L and describe how this affects the algorithm.

The paper defines the last suffix, \( S_{n-1} \), to be both S- and L-type. It is not really clear what is meant by this, but we discovered that in order for the algorithm to work, the sentinel should have the same type as the sample suffixes, i.e. S-type in this case. The reason for this is to ensure the validity of the input string used when calling the algorithm recursively as well as for induced sorting to work. This is argued in more detail in Lemma 6.11 and Lemma 6.13.

#### 6.2.2 Sample sorting

##### 6.2.2.1 S-substrings and their order

We start by defining what the authors refer to as S-substrings:

**Definition 6.4 (S-substrings)**

For a given S-type suffix \( S_i \), the corresponding S-substring is the prefix of \( S_i \) ending at the first S-type character after \( S[i] \). We denote this string as \( \text{sub}(S_i) \).
For convenience we let the length 1 sentinel-suffix be the only length 1 \(S\)-substring.

That is, with the sentinel as the only exception, an \(S\)-substring consists of a single \(S\)-type character followed by zero or more \(L\)-type characters and a single \(S\)-type character. This is illustrated in Figure 5.

![Figure 5: S-substrings illustrated with intervals. Note that neighbouring S-substrings share endpoints.](image)

At a high level, the reason for introducing \(S\)-substrings is that the sum of their lengths is \(\Theta(n)\) in the worst case, whereas the sum of the lengths of the \(S\)-type suffixes is \(\Theta(n^2)\) in the worst case. This will be important for ensuring a linear time complexity – we return to this in section 6.3. Intuitively, we want to sort the \(S\)-substrings in such a way that we can use that sorting to say something about the order of the corresponding \(S\)-type suffixes.

In the original paper, the \(S\)-substrings are sorted lexicographically with one critical exception: \(S\)-substrings that are proper prefixes of other \(S\)-substrings are greater than their superstrings. That is, if \(S\)-substring \(T\) is a prefix of \(S\)-substring \(T^*\), then \(T\) comes after \(T^*\) in ascending sorted order, which is opposite normal lexicographical ordering.

When first reading the paper it was not clear why this prefix exception is needed and it caused some implementation issues, so at this point we divert from the paper by explicitly defining what we will refer to as extended character order and, subsequently, extended string order. This will be used to make it clear why the authors impose the just mentioned prefix constraint.

**Definition 6.5 (Extended character order, \(\leq_E\))**

Consider two characters \(a, b \in \Sigma\) occurring in strings \(S_a\) and \(S_b\) respectively. Let \(t(a)\), \(t(b)\) denote the \(S/L\)-types of \(a\) and \(b\) as defined in Definition 6.1, i.e. based on the types of their corresponding suffixes of \(S_a\) and \(S_b\).

Now, \(a\) is smaller than \(b\) with respect to this extended ordering, which we write \(a <_E b\), if and only if one of the following two conditions is satisfied:

1. \(a < b\)
2. \(a = b \text{ and } t(a) = L \text{ while } t(b) = S\)

If neither \(a <_E b\) nor \(b <_E a\) holds, then \(a\) and \(b\) are equal with respect to this extended character ordering, in which case we use the notation \(a =_E b\). Note that two characters are equal if and only if \(a = b \text{ and } t(a) = t(b)\).
Note that the above definition corresponds to lexicographical order of the tuples \((a, t(a))\) and \((b, t(b))\).

We can now expand this definition to strings:

**Definition 6.6 (Extended string order, \(\leq_E\))**

Consider two strings \(S\) and \(T\). The order of \(S\) and \(T\) with respect to this extended string order (also denoted \(\leq_E\)) is determined as follows:

- \(S\) is smaller than \(T\), which we write \(S <_E T\), if and only if it holds that there exists an index \(i\) such that \(S[i] <_E T[i]\) and \(S[j] =_E T[j]\) for \(j < i\).

If neither \(S <_E T\) nor \(T <_E S\) holds, then \(S\) and \(T\) are equal with respect to this extended string ordering, in which case we use the notation \(S =_E T\).

A set of strings sorted with respect to this extended order are said to be in \(\leq_E\)-order.

Note that this definition implies that strings of unequal length can be equal. However, for our purposes that will never happen and it is therefore not relevant.

The following lemma summarizes why this ordering is useful when relating \(S\)-substrings and \(S\)-type suffixes:

**Lemma 6.7 (\(S\)-substrings, \(S\)-type suffixes and extended string order)**

*For two \(S\)-type suffixes \(S_i\) and \(S_j\) it is the case that if \(\text{sub}(S_i) <_E \text{sub}(S_j)\) then \(S_i < S_j\).*

**Proof.** Assume \(\text{sub}(S_i) <_E \text{sub}(S_j)\). This means that there exists an index \(m\) such that \(\text{sub}(S_i)[m] <_E \text{sub}(S_j)[m]\) and \(\text{sub}(S_i)[l] =_E \text{sub}(S_j)[l]\), for \(l < m\).

By Definition 6.5, the fact that \(\text{sub}(S_i)[l] =_E \text{sub}(S_j)[l]\), for \(l < m\), means that in particular \(\text{sub}(S_i)[l] = \text{sub}(S_j)[l]\), i.e. \(\text{sub}(S_i)\) and \(\text{sub}(S_j)\) share a length \(m\) prefix. Therefore, the lexicographical order of \(S_i\) and \(S_j\) is determined by the order of \(S_{i+m}\) and \(S_{j+m}\). For convenience, let \(c_i = \text{sub}(S_i)[m] = S_{i+m}[0]\) and \(c_j = \text{sub}(S_j)[m] = S_{j+m}[0]\).

Since \(c_i < c_j\), either condition 1 or 2 in Definition 6.5 holds. We handle these two cases separately:

1. In this case \(c_i < c_j\) so clearly \(S_{i+m} < S_{j+m}\) and therefore \(S_i < S_j\).
2. In this case \(c_i = c_j\) but \(t(c_i) = L\) while \(t(c_j) = S\). By definition, this means that \(S_{i+m}\) is an \(L\)-type suffix while \(S_{j+m}\) is an \(S\)-type suffix. It then follows from Lemma 6.2 that \(S_{i+m} < S_{j+m}\) which again implies \(S_i < S_j\).

\(\square\)

What this lemma says is that we can use the order of two \(S\)-substring that are different with respect to \(\leq_E\) to infer the lexicographical order of the corresponding suffixes. We explicitly state the following observations:
Observation 6.8
The implication does not necessarily hold in the other direction. That is, if $\text{sub}(S_i) \leq_E \text{sub}(S_j)$ it might be the case that $S_i < S_j$ or $S_i > S_j$. In other words, if $\text{sub}(S_i) = \text{sub}(S_j)$ we cannot say anything about the order of $S_i$ and $S_j$.

Observation 6.9
Two S-substrings of different length can never be equal with respect to $\leq_E$. Even if an S-substring $T$ is a proper prefix of another S-substring $T^*$ then, because $T[|T| - 1]$ is S-type and $T^*[|T| - 1]$ is L-type, we have $T^* <_E T$.

Observation 6.9 explains why the authors point out that prefix S-substrings should be ranked higher than their superstrings.

6.2.2.2 Sorting the S-substrings
Note that, as with regular lexicographical ordering, the extended order of two strings can be determined by comparing their characters one by one (with respect to $\leq_E$) in a left to right scan. This means the S-substrings can be Radix-sorted. In order to facilitate this we start by defining what the authors refer to as the S-distance of a suffix of $S$:

**Definition 6.10** (S-distance)
The S-distance of suffix $S_i$ is the distance from $S_i$ to the nearest S-type suffix to the left, excluding $S_i$ itself. If there is no S-type suffix to the left of $S_i$ its S-distance is defined to be 0.

Now, let dist be an array of size $n$ and for each suffix $S_i$ in $S$, compute its S-distance and store it in $\text{dist}[i]$. An example of S-distances can be seen in Figure 6.

```
M I I S S I I S S I I P P I I $
S/L:  L S S L S L S L S L S L S
S-dist:  0 0 1 1 2 3 1 2 3 4 1 2 3 4 5 6
```

**Figure 6:** String with corresponding S-distances

In the paper, the authors then define the array $SA_1$ to contain all suffixes of $S$ in order based on the (regular) order of their first character, that is the suffixes are placed somewhere in their suffix array buckets. However, based on the previous discussion about S-substring ordering, we modify this slightly to instead let $SA_1$ contain all suffixes of $S$ in order based on the extended order of their first character. Another way of thinking of this is that the suffixes are placed somewhere in their S-/L-buckets as previously defined in definition 6.3.

Now, let $m$ be the largest S-distance and create a new array $\text{distLists}$ of $m$ lists, such that list $\text{distLists}[i]$, $1 \leq i \leq m$, contains the indices of all suffixes with S-distance $i$, in the order in which they appear in $SA_1$. That is, in each of the $m$ distance lists the suffixes appear in ascending order, sorted on their first character with respect to $\leq_E$. 
Now the S-substrings will be sorted in $\leq E$-order, using the distance lists created above. In order to do this, notice that a suffix, $S_i$, in $\text{distLists}[k]$ has the property that its first character is the $k^{\text{th}}$ character of the S-substring starting at index $i - k$. By processing the distance lists in turn, this can be utilized to Radix-sort the S-substrings.

Let $A_i$ be an array containing the indices of the S-substrings. It has the property that the S-substrings occur in $A_i$ ordered on their length $i$ prefixes with respect to $\leq E$. For an S-substring, $T$, let $I(T)$ denote the interval of $A_i$ that contains the S-substrings that have the same length $i$ prefix as $T$. This is possible because substrings that share a prefix occur in a contiguous part of $A_i$.

Notice that $A_1$ can be computed via a single scan of $SA_1$. Recall that $m$ was the largest S-distance or, equivalently, the length of the longest S-substring. We are interested in computing $A_m$ since this array exactly provides the $\leq E$-order of the S-substrings. In Algorithm 1 we show how to transform $A_{i-1}$ into $A_i$.

```
/* Turns $A_{i-1}$ into $A_i$ in-place */
Function Radix-round($A_{i-1}$, distLists[i]):
  foreach $j \in \text{distLists}[i]$ in order do
    Let $T$ be the S-substring starting at index $j - i$, i.e. $\text{sub}(S_{j-i})$
    Move $T$ to the head of $I(T)$
    Increment the head of $I(T)$

  foreach $j \in \text{distLists}[i]$ in order do
    Let $T$ be the S-substring starting at index $j - i$
    Restore $I(T)$

Algorithm 1: Pseudocode for computing $A_i$ from $A_{i-1}$
```

We want Algorithm 1 to run in time $O(|\text{distLists}[i]|)$. It might not be clear how to do this, especially how to restore the S-substring intervals in Line 9 – in Section 6.5 we describe the way we did it.

Using Algorithm 1, the S-substrings can be sorted in $\leq E$-order by iterating through the distance lists as sketched in Algorithm 2.

Having obtained $A_m$, we let the rank of an S-substring be its order in $A_m$, where S-substrings that share an interval have the same rank, because that means they are equal with respect to $\leq E$. We then build a new string, $S'$, whose characters are the ranks of the S-substrings in the order they appear in $S$. The pseudocode in Algorithm 3 sketches this process.

Once Algorithm 3 is complete, the characters of $S'$ are examined. If these are all unique, then the order of the S-substrings determine the order of the S-type suffixes per lemma 6.7 and we can compute the suffix array $SA'$ of $S'$ directly. Otherwise, at least two S-substrings are equal with respect to $\leq E$ meaning, by Observation 6.8, that the order of at least two sample suffixes is still undetermined. What we
Function Sort-$S$-substrings():
  Initialize $A_i$ to $A_1$ via a scan of $SA_1$
  for $i = 1$ to $m$
    Radix-round($A_i$, distLists[$i + 1$])
  return $A_i$

Algorithm 2: Sorting the $S$-substrings

Function Construct-recursive-input-string($A_m$):
  // Assign ranks to $S$-substrings
  $r \leftarrow 0$
  foreach $S$-substring $T \in A_m$ in order do
    if $T$ does not share interval with the previous $S$-substring then
      $r \leftarrow r + 1$
    Assign rank $r$ to $T$
  // Construct $S'$ using $S$-substring ranks
  Initialize $S'$ to the empty string
  for $i = 0$ to $n - 1$
    if No $S$-substring starts at index $i$ in $S$ then
      Continue to next loop iteration
    Let $r$ be the rank of $S$-substring sub($S_i$)
    Append $r$ to $S'$
  return $S'$

Algorithm 3: Constructing $S'$

then do is compute $SA'$ recursively. $S'$ is a valid input string as we later argue in Lemma 6.11.

The claim is that the order of the suffixes of $S'$ directly provide the order of the sample suffixes of $S$. More precisely, let $r(i)$ be the index of $S'$ where the rank of $S$-substring sub($S_i$) is located. Then for two sample suffixes $S_i$ and $S_j$, the claim is that

$$S_i < S_j \iff S'_{r(i)} < S'_{r(j)}$$

meaning $SA'$ can be used to infer the order of the sample suffixes. We return to this in Lemma 6.12.

6.2.3 Induced sorting

After the sample sorting phase, we know the lexicographical order of the $S$-type suffixes. These can then be placed in their final places in $SA$ by iterating through them in descending order and placing them at the right-most vacant entry of their suffix array buckets. This places the $S$-type suffixes correctly in $SA$, due to $S$-type suffixes appearing after $L$-type suffixes in a given suffix array bucket, as previously shown in Lemma 6.2.
Once the S-type suffixes are placed in $SA$, inducing the order of the remaining suffixes, that is the L-type suffixes, proceeds as follows:

Scan $SA$ from left to right. For each non-empty entry $SA[i]$, if suffix $SA[i] - 1$ is an L-type suffix, put suffix $SA[i] - 1$ at the current front of its suffix array bucket and move the front of the bucket one position to the right.

We return to why this works when arguing that the algorithm is correct in Section 6.4.

### 6.2.4 When sample suffixes are L-type

As mentioned in the introduction of this chapter, when the input string contains fewer L-type than S-type suffixes, the L-type suffixes are used as the sample. The paper only briefly touches on how this affects the algorithm, so we provide a few more details in this section.

Just as S-substrings and S-distance were defined, one can define L-substrings and L-distance completely analogously. Using the same extended string order, $\leq_E$, Lemma 6.7 can be shown similarly where S-substrings are substituted with L-substrings. That is, the algorithm works unchanged when the sample suffixes are L-type.

Note, however, that prefix L-substring are smaller than their superstrings in $\leq_E$-order which is opposite of what was seen for S-substrings in Observation 6.9.

Having sorted the L-type suffixes, they are placed in their final position in $SA$ by scanning them in ascending order and placing them in the left-most vacant entry in their suffix array bucket. This is symmetric to how the S-type suffixes were placed and the correctness of this also follows from Lemma 6.2.

Inducing the order of the S-type suffixes after the L-type suffixes have been placed in $SA$ is symmetric to the procedure for inducing the order of the L-type suffixes:

Scan $SA$ from right to left. For each non-empty entry $SA[i]$, if suffix $SA[i] - 1$ is an S-type suffix, put suffix $SA[i] - 1$ at the current end of its suffix array bucket and move the end of the bucket one position to the left.

### 6.3 Time Complexity

#### 6.3.1 Sample selection

The sample selection simply consists of determining the type of each suffix, and keeping track of how many of each type there are. This can be done in linear time as described in Definition 6.1.
6.3.2 Sample sorting

Determining the $S$-distances can be done in $\Theta(n)$ time, by simply scanning $S$ from left to right and keeping track of the distance to its nearest $S$-type position to the left – determining $m$, i.e. the highest distance seen, can be done during this scan.

Creating the array $\text{distLists}$ can be done by scanning $SA_1$ from left to right and for each $i \in SA_1$ append $\text{dist}[i]$ to $\text{distLists}[\text{dist}[i]]$. Because $|SA_1| = n$, scanning $SA_1$ takes linear time and appending to a list in $\text{distLists}$ can be done in $O(1)$, for instance by using linked lists. Alternatively, arrays can be used instead of linked list by first counting how long each array should be via a scan of $\text{dist}$, which takes an additional $\Theta(n)$ time. In any case, $\text{distLists}$ can be created in $\Theta(n)$ total time.

Sorting the $S$-substrings is done by processing $\text{distLists}[1 \ldots m]$ in turn as described in Algorithm 2. Processing an individual distance list, $\text{distLists}[i]$, as described in Algorithm 1, can be done in time $\Theta(|\text{distLists}[i]|)$ and so the total time for sorting the $S$-substrings is $\Theta(\sum_{i=1}^{m} |\text{distLists}[i]|)$. Recall that this sum is upper bounded by the total size of the $S$-substrings. Since the total size of the $S$-substrings is $O(n)$, so is the sum$^*$. 

Constructing $S'$ is done via one scan of $A_m$ and one scan of $S$, as described in Algorithm 3. The size of $A_m$ is equal to the number of $S$-substrings, of which there are at most $n/2$ because we have assumed there are fewer $S$- than $L$-type suffixes. So the total time for the two scans is $O(n)$.

$S'$ has a character for each $S$-substring so, as was the case for $A_m$, its size is no larger than $n/2$. If every character of $S'$ is unique, we compute $SA'$ directly in linear time via a single scan of $S'$. Otherwise, we recurse.

6.3.3 Induced sorting

The induced sorting can clearly be done in $\Theta(n)$ time, since it just involves a single scan of $SA$ whose size is $n$. The only bookkeeping necessary is to keep track of the current head of each suffix array bucket, which can for instance be done using an array of size $|\Sigma|$. Since $|\Sigma| = O(n)$ and this array is updated $O(n)$ times, each update taking constant time, this bookkeeping does not incur an asymptotic increase in running time.

$^*$It might be $O(1)$ for best-case input. An example is when there is only one sample suffix, because in this case $\sum_{i=1}^{m} |\text{distLists}[i]| = 0$. 
6.3.4 Total time

In total the above three steps take time $\Theta(n)$ for any input, not counting the recursion. If we recurse, recall that $S'$, the recursive input, has size at most $n/2$ for any input, so the solution to the following recurrence is the worst case running time of the algorithm:

$$T(n) = T(n/2) + \Theta(n)$$

This recurrence solves to $T(n) = \Theta(n)$ by the master theorem [14].

6.4 Correctness

In this section we return to the correctness claims made while describing the algorithm.

6.4.1 Sample sorting

**Lemma 6.11** ($S'$ validity)

In the recursive call, the input string $S'$, constructed as described in Section 6.2.2, is a valid input string to the algorithm. Specifically, the characters of $S'$ are linear in $|S'|$ and the last character of $S'$ is unique and smallest in $S'$ meaning it functions as sentinel.

**Proof.** The characters of $S'$ are the ranks of the $S$-substrings of which there are $|S'|$. All ranks are upper bounded by the number of $S$-substrings, i.e. no character of $S'$ is larger than $|S'|$ and so, in particular, all characters of $S'$ are linear in $|S'|$.

Because the sentinel of $S$ is defined to be an $S$-substring and is lexicographically smaller than every other sample suffix, it alone will get the lowest rank among all $S$-substrings. Furthermore, since the sentinel appears last in $S$, this unique and smallest number will appear last in $S'$, thus functioning as sentinel. \qed

**Lemma 6.12** (Inducing sample order from $SA'$)

There is a one-to-one correspondence between the suffixes of $S'$ and sample suffixes of $S$. That is, by sorting $S'$ we can sort the sample suffixes of $S$.

**Proof.** For the case where the sample suffixes are of type $S$, we refer to Lemma 4 on page 206 in [1] since we find that this lemma is sufficiently detailed. A similar proof can be given when the $L$-type suffixes are the sample suffixes as long as one remembers the fact mentioned in Section 6.2.4 that prefix $L$-substrings, unlike prefix $S$-substrings, are smaller than their superstrings in $\preceq_E$-order.

In essence, the proof is an inductive application of Lemma 6.7. \qed
6.4.2 Induced sorting

**Lemma 6.13**
Given that the $S$-type suffixes are the sample suffixes and that these have been placed in their final positions in $SA$, the procedure described in Section 6.2.3 correctly places the $L$-type suffixes in $SA$.

**Proof.** Notice that when the scan reaches $SA[i]$ and element $S_{SA[i−1]}$ is considered, only $L$-type elements are added – i.e. elements that are lexicographically larger than $S_{SA[i]}$. This means that $SA$ stays intact, up to and including entry $SA[i]$. It will be proved that, when the scan reaches $SA[i]$, all $L$-type buckets in $SA[0...i]$ will already be filled with suffixes in order – i.e. when the scan reaches $SA[n−1]$, $SA$ will contain all $L$-type suffixes in sorted order. This is done by induction on $i$:

- **Base case:** There is always at least one $S$-type suffix, namely the sentinel $S_{n−1}$, which means that the smallest suffix of $S$ must be $S$-type and thus $SA[0]$ contains no $L$-type buckets, making the statement trivially true.
- **Inductive case:** Assume all $L$-type buckets in $SA[0...i]$ have been filled with $L$-type suffixes in order. If $SA[i+1]$ is part of an $S$-type bucket, the statement is trivially true. Assume therefore that $SA[i+1]$ is part of an $L$-type bucket. It needs to be proved that

1. $SA[i+1]$ is non-empty, i.e. it contains some value.
2. $SA[i+1]$ is in its sorted position.

We prove these separately:

1. Assume for contradiction that $SA[i+1]$ is empty and let $j$ be the value that is supposed to be in $SA[i+1]$. Since $j$ has not been added, it must be the case that $j+1$ is not in $SA[0...i]$ – if it was, $j$ would have been added when considering $S_{[j+1]}$. Since $S_j$ is $L$-type, it must be the case that $S_j > S_{j+1}$ and thus $j+1$ must appear before $j$ in the suffix array, a contradiction.

2. Assume for contradiction that $SA[i+1]$ is not in its sorted position. This means that there must exist some $S_k$, where $S_k < S_{SA[i+1]}$ and $k \notin SA[0...i]$. Furthermore, the suffix in $SA[i+1]$ must have the same first character as $S_k$, or it would not have been placed in the $L$-type bucket at $SA[i+1]$. Denote the first character of the two suffixes $c$. Let $S_{SA[i+1]} = c\alpha$ and $S_k = c\beta$ and let $S_\alpha$ and $S_\beta$ denote suffix $\alpha$ and $\beta$ in $S$. Since $S_k$ is $L$-type, $\beta < S_k$ and since $S_k < S_{SA[i+1]}$, it must be the case that $\beta < \alpha$. Because $\beta < S_k$ and the correct sorted position of $S_k$ is $SA[i+1]$, $\beta$ must be referenced in $SA[0...i]$. Note that $S_{SA[i+1]} = c\alpha = S_{\alpha−1}$. This means that $\alpha$ must be referenced in $SA[0...i]$, since $S_{SA[i+1]}$ was added to $SA$ when
considering $S_{\alpha-1}$. All of this is summarized in Figure 7. Since both $\alpha$ and $\beta$ are referenced in $SA[0...i]$, they must appear in sorted order per the induction hypothesis. This means that $\beta$ must be referenced before $\alpha$ in $SA$ and hence $S_k = c\beta = S_{\beta-1}$ must have been considered and added to $SA$ before $S_{SA[i+1]}$, i.e. $k$ must appear in $SA[0...i]$, a contradiction.

\begin{center}
<table>
<thead>
<tr>
<th>Condition</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_k &lt; S_{SA[i+1]}$</td>
<td>Since $S_k$ is L-type</td>
</tr>
<tr>
<td>$S_{SA[i+1]} = c\alpha = S_{\alpha-1}$</td>
<td></td>
</tr>
<tr>
<td>$S_k = c\beta = S_{\beta-1}$</td>
<td></td>
</tr>
<tr>
<td>$\beta &lt; S_k$</td>
<td>Since $S_k &lt; S_{SA[i+1]}$</td>
</tr>
<tr>
<td>$\beta &lt; \alpha$</td>
<td>since $\beta &lt; S_k$ and $SA[i+1]$ should be $k$</td>
</tr>
<tr>
<td>$\beta \in SA[0...i]$</td>
<td>Since $S_{SA[i+1]}$ was added to $SA$ when considering $S_{\alpha-1}$</td>
</tr>
<tr>
<td>$\alpha \in SA[0...i]$</td>
<td></td>
</tr>
</tbody>
</table>
\end{center}

Figure 7: Steps of the proof summarized

A symmetric lemma for when the L-type suffixes are the sample suffixes can be proven:

**Lemma 6.14**

*Given that the L-type suffixes are the sample suffixes and that these have been placed in their final positions in $SA$, the procedure described in Section 6.2.4 correctly places the S-type suffixes in $SA$.*

**Proof.** This lemma can be proven in a fashion completely symmetrical to that of the proof of Lemma 6.13 so we have omitted it here.

6.4.3 Summary

**Theorem 6.15** (Correctness of the KA algorithm)

*The KA algorithm correctly computes the suffix array of an input string of length $n$ in time $\Theta(n)$.*

**Proof.** Follows from the time complexity and correctness arguments in Section 6.3 and Section 6.4.

6.5 Implementation Notes

In the following, we use the term sample-substring instead of S- or L-substring because the algorithm works in both cases as discussed in Section 6.2.4.
Creating $A_1$ and the corresponding intervals is, as mentioned in the algorithm description, easily done from $SA_1$.

For us, the main implementation challenge was transforming $A_{i-1}$ to $A_i$ in $O(\|\text{distLists}[i]\|)$ time.

The pseudocode presented in Algorithm 1 reflects our solution which we elaborate on here.

We represent $A_{i-1}$ using an array, $A$ and a simple class $\text{Interval}$ which has two fields, $\text{start}$ and $\text{end}$. The indices of the sample-substrings are stored in $A$ and for each sample-substring, $\text{sub}(S_j)$, an array $I[j]$ refers to an instance of $\text{Interval}$ with the property that $\text{sub}(S_j)$, and the sample-substrings with whom $\text{sub}(S_j)$ shares a length $i$ prefix, are contained in $A[I[j].\text{start}, \ldots, I[j].\text{end}]$. Furthermore, an array $P[j]$ provides the position in $A$ at which $\text{sub}(S_j)$ is stored.

The first loop in Algorithm 1 is quite easy to handle. For some $j \in \text{distLists}[i]$ we swap $A[I[j].\text{start}]$ with $A[P[j]]$. Then $P[j]$ and $P[I[j].\text{start}]$ are updated and $I[j].\text{start}$ is incremented.

After this first loop, the intervals of the sample-substrings in $\text{distLists}[i]$ are no longer valid and the purpose of the second loop in Algorithm 1 is to correct this.

The way we did this was to notice that all sample-substrings that shared an interval before the first loop appear in consecutive entries of $A$ after the first loop. We can use this when looping through the sample-substrings a second time.

For each $j \in \text{distLists}[i]$, check if $P[j] >= I[j].\text{start}$. In this case the interval for $\text{sub}(S_j)$ has already been corrected, so continue to the next sample-suffix. If, on the other hand, it is the case that $P[j] < I[j].\text{start}$, $I[j]$ has not been updated. To do this, create a new instance of $\text{Interval}$ and store this at $I[j]$. Initially, let $I[j].\text{start} = I[j].\text{end} = P[j]$. We now want to correct the intervals of every sample-substring that shared an interval with $\text{sub}(S_j)$ before the first loop. These will occur at entries $A[P[j]+1], A[P[j]+2]$ etc. as noted above.

Say we are looking at $A[P[j]+k]$ and that sample-substring $\text{sub}(S_i)$ is stored here. Whether $\text{sub}(S_i)$ needs updating can be detected by checking whether $P[i] < I[1].\text{start}$. If it does, then we check if $\text{sub}(S_j)[i] = E$ $\text{sub}(S_i)[i]$, i.e. we check if $\text{sub}(S_j)$ and $\text{sub}(S_i)$ share a length $i$ prefix. If they do, we expand $I[j]$ by setting $I[j].\text{end} = P[i]$ so $\text{sub}(S_j)$ and $\text{sub}(S_i)$ now share an interval again. If they do not, we let $I[1]$ be a new instance of $\text{Interval}$ with $I[1].\text{start} = I[1].\text{end} = P[i]$ and continue the scan from here.

The time for moving the sample-substrings in the first loop is clearly $O(\|\text{distLists}[i]\|)$. While correcting intervals in the second loop, we visit each sample-substring in $\text{distLists}[i]$ at most twice and spend

---

*Remember that $\text{sub}(S_j)$ and $\text{sub}(S_i)$ shared an interval and therefore a length $i-1$ prefix before this Radix-sort round, so it’s enough to check the $i$th character.*
constant time at each visit, so the total time for interval correction is also $O(|\text{distLists[i]}|)$ as desired.
THE SKEW ALGORITHM

The Skew algorithm by Kärkkäinen and Sanders [2] is a worst-case linear time suffix array construction algorithm from 2003 like KA. However, its suffix sample criterion and sample sorting technique is radically different. Suffixes are not partitioned based on their contents, as was the case with S- and L-type suffixes, but rather based on their positions in $S$.

7.1 ALGORITHM DESCRIPTION

7.1.1 Sample selection

The Skew algorithm splits the input suffixes into three sets as follows:

Given a string $S$, let $S^0$ be the set consisting of all the suffixes starting at positions $i \in \{0 \ldots n - 1\}$ where $i \mod 3 = 0$. That is, $S^0$ consists of suffixes $S_0, S_3, S_6$ etc. Similarly, let $S^1$ consist of all the suffixes starting at positions $i \in \{0 \ldots n - 1\}$ where $i \mod 3 = 1$, and $S^2$ consist of the suffixes starting at positions where $i \mod 3 = 2$.

Now let $S^{12} = S^1 \cup S^2$. $S^{12}$ is the sample of suffixes that will be sorted and then used to induce the order of the suffixes in $S^0$.

7.1.2 Sample sorting

Given $S^{12}$, the sample sorting part of the Skew algorithm creates an array $SA^{12}$, which is the suffix array for all suffixes in $S^{12}$. This is done as follows.

First the suffixes in $S^{12}$ are sorted based on their 3-grams which are their three first characters. We denote the resulting order as the 3-order of the suffixes. The 3-order can be computed using Radix-sort.

Given the 3-order of the suffixes in $S^{12}$, we assign a rank to each suffix in $S^{12}$. The rank of a suffix is its position in the 3-order. Note that suffixes who share the same first three characters are assigned the same rank. An example of this can be seen in Figure 8. We now have two cases:

---

*What we mean by a “suffix array” of a set of suffixes, like $SA^{12}$, is an array which provides the lexicographical order of the suffixes and thus behaves similarly to a suffix array.

†If the 3-gram of a suffix extends beyond the end of $S$, we pad $S$ with sentinel characters.
• All suffixes in $S^{12}$ have different ranks. In this case the 3-order is equal to the lexicographical order of the suffixes and so we can compute $SA^{12}$ directly, using these ranks.

• Two or more suffixes in $S^{12}$ have the same rank. In this case we create two new strings $A$ and $B$, both initialized to the empty string. We then traverse the suffixes of $S^1 \subseteq S^{12}$ in the order they appear in $S$ and for each suffix append its rank to $A$. Similarly, we traverse the suffixes of $S^2 \subseteq S^{12}$ in the order they appear in $S$ and append their ranks to $B$. Finally, let $S' = A \circ B \circ \$$. Now the algorithm is called recursively with $S'$ as input and the resulting suffix array, $SA'$, is obtained. We show in lemma 7.1 that $S'$ is a valid input string.

We now claim that there is a one-to-one correspondence between $SA'$ and $SA^{12}$, meaning that $SA^{12}$ can easily be obtained from $SA'$ – this claim is proved later in lemma 7.2.

Once $SA^{12}$ has been obtained, the inverse suffix array $ISA^{12}$ can be constructed in linear time.

We now claim that there is a one-to-one correspondence between $SA'$ and $SA^{12}$, meaning that $SA^{12}$ can easily be obtained from $SA'$ – this claim is proved later in lemma 7.2.

Once $SA^{12}$ has been obtained, the inverse suffix array $ISA^{12}$ can be constructed in linear time.

Once $SA^{12}$ has been obtained, the inverse suffix array $ISA^{12}$ can be constructed in linear time.

![Figure 8: Example of suffixes from $S^{12}$ with their corresponding 3-grams and ranks](image)

7.1.3 Induced sorting

Given $SA^{12}$ from the previous step, a new array $SA^0$, which is the suffix array for all suffixes in group $S^0$ will be created. First, all suffixes of $S^0$ are sorted on first character. After this, the ordering of suffixes starting with the same character can be determined using $SA^{12}$. Every suffix in $S^0$ can be written as $S[i] \circ S[i+1]$, where $i+1 \mod 3 \neq 0$. 

![Figure 8: Example of suffixes from $S^{12}$ with their corresponding 3-grams and ranks](image)
This means that the relative ordering of two suffixes from $S^0$ that start with the same character, say suffix $i$ and $j$, is determined by the relative order of two suffixes from $SA^{12}$. We can therefore determine the order of $S_i$ and $S_j$ by looking up $S_{i+1}$ and $S_{j+1}$ in $ISA^{12}$. This means that $SA^0$ is essentially constructed by sorting tuples of the form $(S[i], ISA^{12}[i+1])$ lexicographically$^*$ for $i \mod 3 = 0$. This can be done using Radix-sort.

After $SA^0$ and $SA^{12}$ have been constructed, the only thing left to do is to merge the two into $SA$. This is done similarly to the merge step of the well-known Merge-sort algorithm, that is we consider the suffixes of $S^0$ in sorted order (using $SA^0$) and the suffixes of $S^{12}$ also in sorted order (using $SA^{12}$) and merge the two lists.

Assume the $i-1$ smallest suffixes of $S^0$ and the $j-1$ smallest suffixes of $S^{12}$ have been inserted into $SA$. Let $S_i$ and $S_j$ be the next suffixes in order from $S^0$ and $S^{12}$, respectively. We now want to determine which suffix is smaller and should therefore be inserted into $SA$ next.

Consider the starting letters of $S_i$ and $S_j$. If these do not determine the order between $S_i$ and $S_j$, the order can be determined by considering the following two cases.

- If $j \mod 3 = 1$, consider $S_i = S[i] \circ S_{i+1}$ and $S_j = S[j] \circ S_{j+1}$. Since $i + 1 \mod 3 = 1$ and $j + 1 \mod 3 = 2$, the relative order between $S_{i+1}$ and $S_{j+1}$ can be determined from their relative position in $SA^{12}$, using $ISA^{12}$.

- If $j \mod 3 = 2$, consider $S_i = S[i] \circ S[i+1] \circ S_{i+2}$ and $S_j = S[j] \circ S[j+1] \circ S_{j+2}$. Compare $S[i+1]$ and $S[j+1]$. If this does not determine the order, notice that $i + 2 \mod 3 = 2$ and $j + 2 \mod 3 = 1$, meaning that the relative order between $S_{i+2}$ and $S_{j+2}$ can be determined from their relative position in $SA^{12}$, using $ISA^{12}$.

The above merging of $SA^0$ and $SA^{12}$ will produce the suffix array for the entire string $S$.

### 7.2 Time Complexity

Determining whether a suffix, $S_i$, belongs in $S^0$, $S^1$ or $S^2$ can be done in constant time by computing $i \mod 3$. Therefore, collecting the suffixes of $S^{12}$ can be done in linear time via a single scan of $S$.

In the sample sorting part, the sorting of all elements in $S^{12}$ into 3-order can be done in linear time using Radix-sort, since we sort $\Theta(n)$ strings over a linearly sized alphabet based on only their 3-grams, i.e. a constant number of characters. Assigning ranks to each suffix

$^*$To compare two tuples lexicographically we first compare the first elements and then the second elements if the first elements were equal etc.
in $S^{12}$, and thus also creating $S'$, can be done by scanning through the suffixes in the 3-order provided by the Radix-sort. During this scan, we compare the 3-gram of the current suffix to the 3-gram of the previous, in order to determine if the rank of the current suffix is equal to that of the previous. This comparison takes constant time and is performed once per suffix, so the total time for assigning ranks is $\Theta(n)$. As shown in Lemma 7.2, $SA$ can be obtained from $SA'$ in linear time.

During the induced sorting, $S^0$ can be sorted in linear time using Radix-sort, where the order of two suffixes $S_i, S_j \in S^0$ is determined by comparing the tuples $(S[i], ISA^{12}[i + 1])$ and $(S[j], ISA^{12}[j + 1])$.

During the merging phase, the ordering of two suffixes, $S_i \in SA^0$ and $S_j \in SA^{12}$, can be determined in constant time by inspecting a constant number of symbols and using $ISA^{12}$ when necessary. Since we fill out one entry of $SA$ each time, the entire merging step takes $\Theta(n)$ time.

From the above, it can be seen that the total running time is $\Theta(n)$, plus the time the recursion takes, in the worst case. Since $S'$ contains $\lfloor \frac{2n}{3} \rfloor + 1$ characters, the total running time is a solution to the recurrence relation

$$T(n) = \Theta(n) + T\left( \lfloor \frac{2n}{3} \rfloor + 1 \right)$$

This recurrence solves to $T(n) = \Theta(n)$ by the master theorem [14].

### 7.3 Correctness

In this section we state and prove a few lemmas in significant detail because they are essential for establishing the correctness of the algorithm, but also because they are largely only stated with little to no proof in [2].

#### 7.3.1 Sample sorting

**Lemma 7.1** ($S'$ is a valid input string)

$S'$, constructed as described in Section 7.1.2, is valid input to the Skew algorithm.

**Proof.** In order for $S'$ to be a valid input string, it is required that the characters/integers of $S'$ are linear in $|S'|$. This is required in order for Radix-sort to run in linear time. Since each character of $S'$ is a rank and each rank is in the set $\{0, 1, \ldots, |S^{12}|\}$, this is indeed satisfied because $|S'| = |S^{12}|$.

Furthermore, the last character of $S'$ should be a sentinel, which it is by construction. 

$\square$
7.3 Correctness

7.3.2 Induced sorting

**Lemma 7.2** (Inducing \(SA^{12}\) from \(SA'\))

The suffix array \(SA^{12}\) of \(S^{12}\) can be obtained from the suffix array \(SA'\) of \(S'\) in linear time.

**Proof.** Let \(n_1 = |S^1|\) and \(n_2 = |S^2|\). Recall that \(S'\) consists of the ranks of the suffixes of \(S^1\) in the order they appear in \(S\), followed by the \(S^2\) suffixes, also in the order they appear in \(S\). That is, \(S'\) has the form

\[
S' = A \circ B \circ S
\]

where \(A = a_1 \circ a_2 \circ \ldots \circ a_{n_1}\) and \(B = b_1 \circ b_2 \circ \ldots \circ b_{n_2}\). The \(a_i's\) are the ranks of \(S^1\) suffixes and the \(b_j's\) are the ranks of \(S^2\) suffixes. We first argue that \(a_{n_1}\) and \(b_{n_2}\), the last characters of \(A\) and \(B\), are both unique in \(S'\).

\(a_{n_1}\) is the rank of the suffix in \(S^1\) that occurs last in \(S\). Depending on its position in \(S\), the 3-gram of this suffix contains 1, 2 or 3 sentinel characters and is the only suffix in \(S\) whose 3-gram contains that number of sentinel characters. That means its 3-gram is unique in \(S\) and consequently its rank unique in \(S'\).

Similarly, the 3-gram of the suffix corresponding to \(b_{n_2}\) contains a unique number of sentinel characters, making its 3-gram unique in \(S\) and consequently its rank unique in \(S'\).

Define the function \(r(i)\) to be the position in \(S'\) where the rank of suffix \(S_i\) is located, see Figure 9 for an example. Specifically

\[
r(i) = \begin{cases} 
    \frac{i-1}{3}, & \text{if } S_i \in S^1 \\
    n_1 + \frac{i-2}{3}, & \text{if } S_i \in S^2 
\end{cases}
\]

Note that 3 evenly divides into \(i - 1\) in the first case, because \(i \mod 3 = 1\), when \(S_i\) is in \(S^1\). For similar reasons, 3 evenly divides \(i - 2\) in the second case. Also, because \(r\) is a permutation, \(r^{-1}\) exists and can furthermore easily be computed in \(O(1)\) time.

Now consider two suffixes \(S_i, S_j \in S^{12}\). The claim is now that

\[
S_i < S_j \iff S'_{r(i)} < S'_{r(j)}
\]  

(1)

The reason this holds is that we can compare \(S_i\) and \(S_j\) by first comparing the strings \(S[i] \circ S[i+1] \circ S[i+2]\) and \(S[j] \circ S[j+1] \circ S[j+2]\), i.e. the first three characters of each suffix. But because the characters of \(S'\) are ranks of the \(S^{12}\) suffixes based on their 3-order, this is equivalent to comparing the ranks \(S'[r(i)]\) and \(S'[r(j)]\). If this comparison does not determine the order of \(S_i\) and \(S_j\), we can compare the next three characters which, because suffixes in \(S^1\) (and also \(S^2\)) are offset by three characters, is equivalent to comparing \(S[r(i+1)]\) to \(S[r(j+1)]\). This can be continued until the ranks differ, determining the order. Note that because \(a_{n_1}\) and \(b_{n_2}\), as previously shown, are
both unique, the order of two suffixes will always be resolved when either of these ranks is reached, or earlier.

This exactly means that comparing the $i^{\text{th}}$ and $j^{\text{th}}$ suffixes of $S$ is equivalent to comparing the $r(i)^{\text{th}}$ and $r(j)^{\text{th}}$ suffixes of $S'$.

Finally, (1) gives us that $SA'_{12}$ can be constructed by scanning $SA'_{12}$ from left to right setting $SA'_{12}[i] = r^{-1}(SA'[i])$, which takes linear time, because we make $O(n)$ computations, each of which take $O(1)$ time.

\section{Implementation Notes}

Radix-sort is at the core of the Skew algorithm since it is used both when sorting the 3-grams and when inducing $SA^0$ from $SA^{12}$. We can think of these two cases as sorting tuples of length 3 and 2, respectively. Due to its importance, the purpose of this section is to elaborate on how we implemented it. The authors of the original paper suggest an implementation of Radix-sort that doesn’t allocate new

\begin{footnote}
In other words, for any suffix in $S^{12}$, its order is always determined using only characters from A or only characters from B. See Figure 9 for an example.
\end{footnote}
arrays but instead reuses two arrays. Our Radix-sort implementation closely follows this suggestion with only minor modifications.

In the Skew algorithm, only tuples of equal length are sorted, which simplifies the implementation, since no padding or other measures are needed. Radix-sort implementations can be divided into two main categories, namely those that start from the most significant digit (MSD), or left-most tuple entry in our case, and those that start from the least significant digit (LSD), or right-most tuple entry. Our Radix-sort implementation is of the LSD variant and is implemented as a series of Counting-sort passes. The Counting-sort procedure has the following signature:

```java
void countingSort(int[] input, int[] outputArray, int[] indexArray, int offset, int maxElementValue)
```

The `input` array consists of \( n \) values that will be used as sorting keys. The array `indexArray` consists of indices, each in the interval \([0, 1, \ldots, n - 1]\). Each index represents a tuple and their order in `indexArray` reflects the current order of the tuples. During the current round of Counting-sort, a tuple, represented by index \( i \), will be sorted with respect to the value `input[i + offset]`. The array `outputArray` is of the same size as `indexArray` and is where the indices of the tuples will be put in their new, updated order after this Counting-sort pass. Finally, `maxElementValue` is the maximum value in `input` and can as such be thought of as the alphabet size.

The overall steps of the Counting-sort procedure are as follows:

1. Create a counting array, `int[] c`, of size `maxElementValue`.
2. Count how many times each key occurs, i.e. iterate through `indexArray` and for each index \( i \), increment `c[input[i + offset]]`.
3. Then transform `c` to contain the prefix sums of the counts, i.e. instead of `c[k]` being the number of keys equal to \( k \), it is instead the number of keys strictly smaller than \( k \). For instance, for the smallest key, \( k' \), this means `c[k']` is 0.
4. Having computed the prefix sums, note that tuples with key \( k \) should appear in succession in `outputArray`, starting from index `c[k]`. So we iterate through `indexArray` from left to right and for each index \( i \), let \( k = input[i + offset] \). Then set `outputArray[c[k]] = i`. Finally, increment `c[k]` so the current entry in `outputArray` is not overwritten by the next tuple with key \( k \).

A crucial requirement for this Counting-sort procedure to work in a LSD Radix-sort is that it is stable, i.e. that the current order of elements that have equal keys is preserved. This is ensured in step

*Due to the offset, care has to be taken to ensure that the bounds of `input` are not exceeded. In our case the offset is always at most 2, so we padded `input` with two extra sentinels.*
by iterating through indexArray from left to right and placing the
indices in outputArray, also from left to right.

Because this Counting-sort procedure updates the order in indexArray
and puts the new order in outputArray, outputArray can be reused as
indexArray in a new round of Counting-sort. As an example of this,
the following snippet shows how we sort the 3-grams of the suffixes
in $S^{12}$:

```
// Initially workArray contains the indices of
// the $S^{12}$ suffixes in any order

// Sort on least significant character in 3-gram
countingSort(input, mod1Or2SA, workArray, 2, alphabetSize);
// Then sort on next, more significant character
countingSort(input, workArray, mod1Or2SA, 1, alphabetSize);
// And finally most significant character in 3-gram
countingSort(input, mod1Or2SA, workArray, 0, alphabetSize);
```

Listing 1: Sorting 3-grams using three passes of Counting-sort.

Notice how the two arrays workArray and mod1Or2SA are reused be-
tween Counting-sort passes and take turns storing the current order
of the 3-grams. mod1Or2SA stores the final, 3-gram based order of the
$S^{12}$ suffixes.
THE SA-IS ALGORITHM

The SA-IS (Suffix Array, Induced Sorting) algorithm made by Nong, Zhang, and Chan [3] is the final worst-case linear time suffix array construction algorithm we consider. It is from 2009 and the most recent of the algorithms treated in this thesis. Like the other algorithms we consider, SA-IS sorts a subset of the suffixes of the input string and then uses that sorting to induce the order of the remaining suffixes. However, unlike any of the other algorithms, SA-IS uses the insight that the technique used to induce the order of the non-sample suffixes can also be used to sort the sample suffixes themselves. As such, the algorithm uses the induced sorting technique almost exclusively, hence the name of the algorithm.

The SA-IS algorithm builds on the ideas introduced in the KA algorithm but replaces the rather involved sample-substring sorting procedure of the KA algorithm, resulting in an arguably more elegant and, based on our implementations, generally faster algorithm which we return to in Part iii.

8.1 Preliminaries

The SA-IS algorithm reuses some of the definitions found in the KA algorithm chapter (Chapter 6), such as S- and L-type suffixes as well as S- and L-type buckets. It does, however, introduce some new definitions which we present in this section.

8.1.1 LMS-characters, -substrings and -suffixes

Definition 8.1 (LMS-character)
Let a character $S[i], i \in \{1 \ldots n-1\}$ be defined as a LMS character (leftmost S), if $S[i]$ is S-type and $S[i-1]$ is L-type.

Having defined LMS-characters we can define LMS-substrings as follows:

Definition 8.2 (LMS-substring)
Let a substring $S[i \ldots j]$, where both $S[i]$ and $S[j]$ are LMS characters and there are no other LMS characters in the substring, be defined as an LMS-substring.

As a special case, the sentinel itself is also defined to be an LMS-substring.

Finally, we define LMS-suffixes:
**Definition 8.3 (LMS-suffix)**

Given a suffix $S_i$, let it be defined as an LMS-suffix, if it is the case that $S[i]$ is an LMS-character.

A string and its corresponding LMS-characters, -substrings and -suffixes can be seen in Figure 10.

![Figure 10: Example of LMS-characters, -substrings and -suffixes](image)

**8.2 Algorithm Description**

Nong, Zhang, and Chan [3] found that instead of sorting all $S$- or $L$-type suffixes of a given string as in the KA algorithm, it suffices to sort all LMS-suffixes. The advantage of this is that there are never more LMS-suffixes than $S$- or $L$-type suffixes meaning a smaller subset of the input string needs to be sorted, before the order of all suffixes of the input can be induced.

**8.2.1 Sample selection**

Given a string $S$, the SA-IS algorithm defines the last suffix, $S_{n-1}$, to be $S$-type and then computes the $S/L$-type of all other suffixes.

For sample selection, all the LMS-suffixes are chosen. As was the case in the KA algorithm, the intuitive reason for introducing LMS-substrings is that their total length is $\Theta(n)$ in the worst case whereas the total length of the LMS-suffixes is $\Theta(n^2)$ in the worst case. We want to order the LMS-substrings in such a way that this order can be used to say something about the order of the corresponding LMS-suffixes.

Unlike the authors of the KA algorithm, the authors of the SA-IS algorithm explicitly state how LMS-substrings should be ordered for them to reflect the order of the corresponding LMS-suffixes. This order is equivalent to the extended string order, $\leq_E$, that was implied in the KA paper and which we explicitly stated in Definition 6.6 in the KA algorithm chapter.

Note that a given string $S$ with length $n$ can have at most $\lfloor n/2 \rfloor$ LMS-characters and hence at most $\lfloor n/2 \rfloor$ LMS-suffixes. This is due to the fact that the first character in the string cannot be an LMS-
character, as it does not have an L-type character to its left, and no two consecutive characters in $S$ can both be LMS-characters.

### 8.2.2 Sample sorting

Given all LMS-substrings from $S$, sample sorting works as follows. Create an array $SA$ that has size equal to the length of $S$. Find the indexes of all buckets in $SA$, i.e. the start and end of all S-type and L-type buckets. The sample sorting then performs the following three steps.

1. Put the index of each LMS-suffix into its S-type bucket in $SA$, according to its first character. Note that the ordering does not matter, i.e. the suffixes belonging to a bucket can be put into that bucket in any order.

2. Scan $SA$ from left to right. For each entry, $SA[i]$, if $S[SA[i]−1]$ is L-type, put $SA[i]−1$ in the current head of its L-type bucket and advance the current head by one.

3. Scan $SA$ from right to left. For each entry, $SA[i]$, if $S[SA[i]−1]$ is S-type, put $SA[i]−1$ in the current end of its S-type bucket and move the current end of the bucket one position left.

After the above steps are done, the LMS-suffixes might have been moved from their original bucketing but the claim is that the LMS-substrings now appear in order in $SA$ with respect to $\leq_E$. We return to this in Lemma 8.4.

We now assign a rank to each LMS-substring based on its position in $SA$. Let $j = 0$. Scan $SA$ from left to right and for each entry $SA[i]$, check if $S[SA[i]]$ is an LMS-substring. While scanning, keep track of the last seen LMS-substring. If $S_{SA[i]}$ is an LMS-suffix, check if the corresponding LMS-substring is equal to the last seen LMS-substring. If this is not the case, let $j = j + 1$. Then let $j$ be the rank of $S_{SA[i]}$.

Now create a new, empty string $S'$. Scan $S$ from left to right, and if $S[i]$ is an LMS-character, append the rank of the corresponding LMS-substring to $S'$.

After scanning all of $S$, $S'$ will be a string of numbers. Create a new array $SA'$ that will be the suffix array of $S'$. If all numbers in $S'$ are unique, $SA'$ can be computed directly from $S'$ by letting $SA'[S'[i]] = i$. If not all numbers in $S'$ are unique, call the algorithm recursively with $S'$ as input to obtain $SA'$. In Lemma 8.5 we briefly argue that $S'$ is a valid input string.

Letting $r(i)$ denote the index of $S'$ where the rank of the LMS-substring starting at $S[i]$ is placed, the claim is, similar to what we saw in the KA chapter, that

$$S_i < S_j \iff S'_{r(i)} < S'_{r(j)}$$
That is, \( SA' \) can be used to directly infer the order of the LMS-suffixes. The argument that this is true is very similar to Lemma 6.12.

### 8.2.3 Induced sorting

The induced sorting is very similar to sample sorting. Given \( SA' \), the induced sorting performs the three steps from sample sorting, where the first step is replaced by the following:

- Put each item in \( SA' \) into its corresponding \( S \)-type bucket in \( SA \), with their relative order preserved as in \( SA' \).

After this, \( SA \) will be the suffix array of \( S \).

### 8.3 Time complexity

The \( S-/L \)-types of each character in \( S \) can be determined in \( \Theta(n) \) time, as described in Definition 6.1. Since each LMS-substring starts and ends with an LMS-character, the LMS-substrings can be found while finding the \( S/L \)-types, by simply keeping track of where the LMS-characters are.

Finding the buckets used in sample sorting can be done in linear time by scanning \( S \) once, counting how many of each character there are and how many of each character has type \( S \) and how many have type \( L \). Counting the characters can be done using an array of size \( |\Sigma| \) because \( |\Sigma| = O(n) \).

Step 1 in sample sorting takes time linear in the number of LMS-suffixes. Step 2 consists of scanning \( SA \) once and doing a constant amount of work for each entry, making it take linear time. Similarly, step 3 takes linear time, as this also consists of scanning \( SA \) once and doing a constant amount of work for each entry. Finally, creating \( S' \) takes linear time – linear time to scan \( SA \), and at most a linear amount of time spent comparing LMS-substrings, as the total length of all LMS-substrings will be linear in the size of \( S \) and each LMS-substring is scanned at most twice. Creating \( SA' \) either takes linear time by scanning \( S' \) once if the characters of \( S' \) are unique, or the time required for the recursion on \( S' \) otherwise.

The induced sorting takes linear time, since step one can be done in time linear in the size of \( SA' \), which has size less than or equal to \( n/2 \).

Given the above, the algorithm has a total running time of \( \Theta(n) + \) “recursion time”. Not counting overlapping characters, every LMS-substring contains at least two characters and so \( S \) can contain at most \( \lfloor n/2 \rfloor \) LMS-substrings/-suffixes for any input. This means that

---

*Not including the sentinel which is a special case LMS-substring of length one.*
the length of $S'$ is at most $|n/2|$ and so the running time of the algorithm on any input is a solution to the recurrence

$$T(n) = \Theta(n) + T(\lfloor n/2 \rfloor)$$

This recurrence solves to $T(n) = \Theta(n)$ by the master theorem [14].

### 8.4 Correctness

#### 8.4.1 Sample sorting

**Lemma 8.4 (LMS-substring sorting)**
The procedure described in Section 8.2.2 correctly sorts the LMS-substrings with respect to $\leq_E$.

*Proof.* This lemma is a corollary of Theorem 2.1 from the original paper. We find that this proof is sufficiently thorough and so refer the reader directly to this. \hfill \square

**Lemma 8.5 (S’ validity)**

$S'$, constructed as described in Section 8.2.2, as a valid input string to the SA-IS algorithm. In particular, its last character is unique and smallest among all characters of $S'$, and the alphabet size/maximum character in $S'$ is linear in $|S'|$.

*Proof.* The sentinel is itself an LMS-substring. It is therefore unique among all LMS-substrings and must have the lowest rank, $j$. Because it occurs last in $S$, $j$ will occur last in $S'$ thus functioning as sentinel in $S'$.

Every character of $S'$ is an LMS-substring rank and is therefore upper bounded by the number of LMS-substrings. The number of LMS-substrings is exactly $|S'|$ and so no character of $S'$ is greater than $|S'|$. In particular, every character is linear in $|S'|$. \hfill \square

#### 8.4.2 Induced sorting

We want to show that after step 2 of the induced sorting procedure, all L-type suffixes are in their correct position in SA.

**Lemma 8.6**

*Having placed the LMS-suffixes in sorted order in SA, the L-type suffixes can be placed in their final positions in SA as described in Section 8.2.3.*

*Proof.* Notice that because we for each entry $SA[i]$ only do something if $S_{SA[i-1]}$ is an L-type suffix, all S-type suffixes that are not also LMS-suffixes are not used when placing the L-type suffixes in SA. This means that the proof of Lemma 6.13, which showed how the order of the L-type suffixes can be induced from the order of the S-type suffixes, works for LMS-suffixes as well. \hfill \square
Once the L-type suffixes are correctly placed in $SA$, the way the remaining S-type suffixes are placed in $SA$ in step 3 is identical to how the KA algorithm does it, and so the correctness of this step follows directly from Lemma 6.14.

8.4.3 Summary

**Theorem 8.7 (Correctness of the SA-IS algorithm)**

The SA-IS algorithm correctly computes the suffix array of an input string of length $n$ in time $\Theta(n)$.

**Proof.** Follows from the time complexity and correctness arguments in Section 8.3 and Section 8.4. 

8.5 Implementation Notes

A major advantage of the SA-IS algorithm is its implementation simplicity. Having found the LMS-suffixes, it essentially boils down to keeping track of the current heads and ends of the suffix array buckets as the suffixes are placed in their final positions in $SA$. As such, we did not encounter any significant issues while implementing the algorithm.
In 2006 Maniscalco and Puglisi [4] published a suffix array construction algorithm (henceforth referred to as the MP algorithm) with the intention that the algorithm should be fast in practice, i.e. fast on data typically encountered in the real world. The trade-off, so to speak, is that the MP algorithm does not provide the same worst-case asymptotic bounds as the other algorithms.

While the MP algorithm does use induced sorting, it differs from the previous algorithms in that it does not utilize the fact that the sample strings it sorts are all suffixes of the same string. Instead, at the core of the algorithm is a sorting procedure specialized for string sorting called Multi-key Quicksort which can sort arbitrary strings. That is, its correctness is not predicated on the input strings being suffixes of the same string.

Primarily due to time constraints, we have implemented the most basic version of the MP algorithm. The paper describes a couple of heuristics whose purpose it is to improve the performance of the MP algorithm, especially on degenerate input. These heuristics are not part of our implementation. However, as we shall see in Part iii, even this basic implementation of the MP algorithm performs very well on certain types of input, including real world input.

9.1 ALGORITHM DESCRIPTION

At the core of the MP algorithm is an efficient string sorting algorithm, Multi-key Quicksort, which was first described in 1997 [15]. While this sorting procedure works well for many strings, it is particularly sensitive to strings that contain a significant number of repeats, i.e. substrings that are repeated throughout a given string. We return to this in Section 9.2.

In this section we describe the algorithm in its most basic, functioning form which uses plain MKQS for sorting the sample suffixes. In Section 9.4 we return to some of the techniques the authors employ in order to improve the performance of the algorithm further.

9.1.1 Suffix sample selection

Exactly as with the SA-IS and KA algorithms, the sample suffixes in MP are selected by first classifying each suffix as S- or L-type. Then, if there are fewer S- than L-type suffixes, the sample suffixes are all the left-most S-type suffixes (LMS-suffixes, as previously described in
Section 8.1.1) – otherwise it’s all the right-most L-type suffixes (RML-suffixes) whose definition is completely symmetric to LMS-suffixes. The sentinel suffix has the same type as the sample suffixes for the same reasons as described in Section 6.2.1, i.e. for induced sorting to work correctly.

9.1.2 Sample sorting

After the sample selection, the sample suffixes are sorted using the previously mentioned string sorting procedure, Multi-key Quicksort, henceforth referred to as MKQS. MKQS borrows and combines ideas from the well-known Quicksort and Radix-sort algorithms. Specifically, from Quick sort it borrows the idea of partitioning the strings based on a pivot value after which the partitions are sorted recursively. From Radix-sort it borrows the idea of sorting on one character/digit at a time, iteratively building an ordering based on the first \((d + 1)\) characters from an ordering based on the first \(d\) characters.

9.1.2.1 MKQS

To be more precise, given a list of strings, \(L\), sorted on the first \(d - 1\) characters, MKQS chooses a pivot character, \(p\), and partitions \(L\) around \(p\) by comparing \(p\) to the \(d\)th character of the strings in \(L\). However, unlike the standard Quicksort algorithm, MKQS creates three partitions instead of two, namely:

- \(L_\prec\): this partition contains the strings \(\{T \in L \mid T[d] < p\}\), i.e. the strings of \(L\) whose \(d\)th character is less than \(p\).
- \(L_\succ\): this partition contains the strings \(\{T \in L \mid T[d] > p\}\).
- \(L_=\): this partition contains the strings \(\{T \in L \mid T[d] = p\}\).

The algorithm is then called recursively on each of the partitions. Since the strings in \(L_=\) are all equal up to and including their \(d\)th character, they are sorted on character \(d + 1\) in the recursive call. The strings in \(L_\prec\) and \(L_\succ\), however, are still only guaranteed to be sorted on their first \(d - 1\) characters, so in the recursive call they are still sorted on their \(d\)th character. Finally, the sorted order of \(L\) is then the sorted order of \(L_\prec\) followed by the sorted order of \(L_=\) and \(L_\succ\).

The recursion stops, when the input list \(L\) has size \(\leq 1\), or when all strings in \(L\) have length \(\leq d\). Pseudocode of this procedure can be seen in Algorithm 4.

The initial input to Algorithm 4 is the list of strings to be sorted and \(d = 0\), that is the strings are sorted starting from their first character. Let \(k\) denote the length of the longest string in \(L\). Note that if all strings in \(L\) do not have the same length, all strings with length less
Algorithm 4: Pseudocode for Multi-key Quicksort

Sample sorting using MKQS

Sample sorting is done simply by using MKQS on the sample suffixes. An initial bucket sort can be performed to place the suffixes in their suffix array buckets (based on their first character) after which each of these buckets can be sorted in turn using MKQS with \(d = 1\).

Induced sorting

Once the sample suffixes have been sorted, the order of the remaining suffixes can be induced exactly as in SA-IS, since the two algorithms use exactly the same sample suffixes in the case where the MP algorithm uses LMS-suffixes as its sample suffixes. In the other possible case, where RML-suffixes are used as sample suffixes, the order of the remaining suffixes can be induced in a symmetric fashion.

9.2 Time complexity

The authors of the algorithm explicitly say in their paper that they have stayed clear of asymptotic time bounds (p. 12 in [4]) because of some of the improvements they make to MKQS.

What we do in this section is provide our own worst- and best-case analysis of the simplified version of the MP algorithm just described, i.e. where the sample suffixes are sorted using basic MKQS.
9.2.1 Worst case

While the MP algorithm, even in its basic form, generally works well in practice as we demonstrate in Chapter 12, some strings can result in immense slowdowns. For instance, consider a highly repetitive string consisting of two different, alternating characters. That is, a string of the form $abab\ldots abab\$. Assume without loss of generality that $a < b$. Then all the suffixes starting with $a$ will be type $S$ suffixes and all the suffixes starting with $b$ will be type $L$ suffixes. This means that the sample suffixes will be every other suffix, say those starting with an $a$ character.

Sorting the sample suffixes with MKQS results in all $n/2$ sample suffixes, except the sentinel, ending up in $L_\leq$ at the top level of the recursion. At the next level all $n/2 - 1$ sample suffixes will end up in $L_\leq$. In general, the input size decreases by one in every other recursive call meaning that the recursion depth becomes $n$. At the $i^{th}$ level of the recursion (counting from the bottom of the recursion tree), the number of suffixes processed is $\lfloor (i + 2)/2 \rfloor$ and so the total number of suffixes processed is

$$\sum_{i=1}^{n} \lfloor (i + 2)/2 \rfloor$$

Because $i/2 \leq \lfloor (i + 2)/2 \rfloor \leq i + 2$ and

$$\sum_{i=1}^{n} i/2 = \Theta(n^2)$$

$$\sum_{i=1}^{n} i + 2 = \Theta(n^2)$$

the total number of suffixes processed is $\Theta(n^2)$.

Finally, each suffix is processed in constant time, and so the total running time becomes $\Theta(n^2)$. Note that this case corresponds to sorting the sample suffixes using only Radix-sort.

The time to induce the order of the remaining $n/2$ non-sample suffixes is $\Theta(n)$ as in SA-IS, so the $\Theta(n^2)$ time for MKQS dominates the running time of the MP algorithm in this case.

9.2.2 Best case

9.2.2.1 MKQS in isolation

Consider a string with no repeated characters. Furthermore, assume that at each level of the recursion with input $(L, d)$ the pivot is the median of $\{S_i[d] \mid S_i \in L\}$. This results in roughly half of the input suffixes ending up in $L_<$ while the other half ends up in $L_>$. The effect is the recursion tree having depth $\Theta(\log(n))$ with $\Theta(n)$ total
work being performed at each level resulting in a total running time of $\Theta(n \log n)$ for MKQS.

Note that this case corresponds to the best case for standard Quick-sort since we never recurse on the $L_<$ partition thus never calling MKQS with $d > 0$.

### 9.2.2.2 MKQS used for sample sorting in the MP algorithm

If there are $\Theta(n)$ sample suffixes then, in the best case, the time for sample sorting using MKQS is $\Theta(n \log n)$ and the time for induced sorting is $\Theta(n)$, so MKQS still dominates the running time.

However, if there are $O(1)$ sample suffixes the time for MKQS is $O(n)\$, meaning induced sorting now dominates and so the total time becomes $\Theta(n)$. This means that although a string of no repeated characters results in the optimal asymptotic running time for MKQS it would still be better to avoid MKQS as much as possible. For instance, although a string of the form $aaa\ldots aaa$ would result in $\Theta(n^2)$ running time if all suffixes were sorted using MKQS alone, such a string contains only one $S$-type suffix (the sentinel) and so it results in a $\Theta(n)$ running time for the MP algorithm.

### 9.3 Correctness

#### 9.3.1 Sample sorting

Given that MKQS is a hybrid of Radix-sort and Quicksort its correctness essentially follows from the correctness of Radix-sort and Quicksort.

To be more specific, we have to argue that if MKQS is called with inputs $L$ and $d$, where all strings in $L$ have equal prefixes of length $d$, then MKQS sorts $L$ correctly.

When MKQS partitions the input strings into $L_<$ and $L_>$ based on their $d^{th}$ character, the order of the suffixes in $L_\leq \cup L_>$ will be correct relative to those in $L_\leq$ because all strings share the same length $d$ prefix. This argument is similar to the argument for why standard Quicksort works.

The strings in $L_\leq$ have equal prefixes of length $d + 1$ and so to determine their lexicographical order, their $(d + 2)^{th}$ character has to be considered which is why MKQS is called with inputs $L_\leq$ and $d + 1$. This is similar to how buckets in standard integer MSD Radix-sort are sorted by looking at the next digit of the integers in a given bucket in which all integers share the same prefix.

---

\*For a case of a time complexity of $\Theta(n)$ with $O(1)$ sample suffixes consider e.g. two sample suffixes of length $\Theta(n)$ with identical characters (up to the suffix). Then the time for MKQS becomes $\Theta(n)$.

\*$d + 1$ and not $d + 2$ because the strings are $0$-indexed.
Finally, note that in the recursive calls of MKQS on $L_<$ and $L_>$ it is always the case that the number of input strings is strictly smaller than $|L|$ because the size of $L_<$ is always at least 1. In the recursive call on $L_=$ the string are sorted on their $(d+1)^{th}$ instead of their $d^{th}$ character. This guarantees that in both cases the base case will be reached eventually meaning that MKQS always terminates.

### 9.3.2 Induced sorting

Since induced sorting is exactly like that of SA-IS, the correctness follows directly from the correctness of SA-IS’s induced sorting, described in Lemma 8.6.

### 9.4 Further Improvements

#### 9.4.1 Improving average and worst case performance

A common strategy for improving standard Quicksort is to switch to Insertion-sort for small input sizes, because the overhead for Insertion-sort is very low due to its simplicity. The MP algorithm authors use the same strategy for MKQS and switch to Insertion-sort when the input size is 64 or less.

In 1997 Musser [16] introduced another, now common, strategy for improving the worst-case performance of standard Quicksort. The observation is that a large recursion depth in Quicksort is a sign that the input is degenerate and might result in the worst-case $\Theta(n^2)$ time complexity for Quicksort. The idea is then to switch to Heap-sort at some pre-determined recursion depth which will result in the worst-case time complexity of $\Theta(n \log n)$ from Heap-sort. The authors of the MP algorithm use the same idea for MQKS and switch to Heap-sort if the recursion depth exceeds 48.

Both Insertion- and Heap-sort for strings can take advantage of their input strings having equal length $k$ prefixes, for some $k \geq 0$, in that string comparisons can start from the $(k + 1)^{th}$ character.

#### 9.4.2 Detecting repetitions

The authors of the MP algorithm recognize that repetitive strings are highly problematic as we demonstrated in Section 9.2.1. As a result, they present an efficient heuristic for detecting and handling potential repetitions. However, we have not included this heuristic in our implementation of the MP algorithm and as such, we will not consider it in further detail.
9.4.3 Utilizing small alphabets

Many strings encountered in practice are built from quite small alphabet such as ASCII [17] which consists of at most 256 characters, which means a character can be encoded using only 2 byte. As such, on a typical machine with 32- or 64-bit words, 4 to 8 characters can fit in one word. One way this can be utilized is that 4 to 8 characters can be compared in the time it takes to compare two machine words.

Another way the MP authors use that $|\Sigma| \leq 256$ is to do an initial bucket sorting of the sample suffixes based on their first two characters instead of only the first. This can be done via a counting sort using relatively small arrays of size $256^2 = 65536$. Then, once the sample suffixes are bucketed based on their first two characters, MKQS can be run on each bucket in turn with $d = 2$.

We do not use this optimization in our implementation. While this hurts performance for small alphabet it has the advantage that our implementation works for linear sized alphabets.

9.5 IMPLEMENTATION NOTES

We implemented the basic version of the MP algorithm described in this chapter, including switching to insertion sort for small input sizes as well switching to a non-recursive Heap-sort implementation once a given recursion depth is reached. The switch to Heap-sort was necessary in order to avoid a stack overflow when sorting degenerate input strings such as strings with many repetitions which lead to linear recursion depth.

Furthermore, we always use the LMS-suffixes as sample suffixes, regardless of whether there are fewer S- or L-type suffixes. Note that, as argued in the SA-IS chapter, the sample contains at most half the suffixes.
Part III

EXPERIMENTAL EVALUATION

In this part, our implementations of the algorithms described in Part ii are tested through various experiments, both individually and against one another. Furthermore, the algorithms are tested on both artificial data and on real world data.

For the artificial data, we have constructed a number of input generators to generate different types of input that will test the strengths and weaknesses of each algorithm.

For the real world data, we have used a collection of DNA sequences that have been widely used to test the performance of suffix array construction algorithms in the past.

Finally, we summarize and discuss the results of the experiments. This includes a discussion of general trends, i.e. which algorithms perform the best, as well as a discussion of performance relative to how hard or easy a given algorithm is to implement.
In this section, we describe how the experiments were conducted. All experiments were run on a machine with an Intel Core i5-2500K and 16GB RAM, running a fresh Ubuntu 17.10 installation.

10.1 WARM UP

Before measuring the running time of an experiment, a number of warm up rounds are executed. The goal of this is to exercise Java’s just in time compiler (JIT), so that all optimizations are in place before starting the actual timing. These optimizations include, but are not limited to, null check elimination, branch prediction, loop unrolling and inlining methods. If these warm up rounds were not executed, there is a chance that one or more optimizations would kick in after a number of runs of a given algorithm, resulting in timings that are uncomparable and likely have a large deviation. We settled on 5,000 warm up rounds, as this was at least as great as the recommendations we found, though we have no source that this is what should be used. Throughout the experiments, we saw no anomalies that we would suspect to be due to JIT optimizations.

10.2 GARBAGE COLLECTION

Whenever we time one of the SACAs we make sure to explicitly do garbage collection by calling System.gc() before the experiment is started. This ensures that garbage collection of unused memory from previous experiments affects the timing of the current experiment as little as possible.

10.3 TIMING AND PLOTTING

For every experiment, each algorithm has been run five times. When plotting the results of an experiment, the average of the five runs is plotted, and the standard deviation is shown as a shadow around each measurement.
The point of this chapter is to describe and demonstrate how different classes of data affect the actual execution time of the algorithms that were described in the second part of the thesis. We discuss our observations, in particular whether these are in line with our expectations.

11.1 ABOUT THE DATA

We constructed a number of different strings to use as input. When referring to an input type, we will use the names given here. The structure of these are described below – note that the sentinel is omitted for readability.

**Heterogeneous:** A string consisting of all different characters in random order, i.e. a string $S$ of length $n$ on the form $S = s_0 \circ s_1 \circ \ldots \circ s_{n-1}$, where $s_i \neq s_j$ for $i \neq j$. Note that, since all characters are different, the alphabet size is equal to the size of the string, i.e. the alphabet size is linear in the length of the input.

**Homogeneous:** A string consisting of identical characters, i.e. a string on the form $aa \ldots a$. This mean the alphabet has size 1.

**Random:** A string consisting of characters chosen randomly from an alphabet consisting of four characters.

**Repetitive:** A string consisting of the same $k$ characters repeated, i.e. a string on the form $[s_1 \circ s_2 \circ \ldots \circ s_k] \circ [s_1 \circ s_2 \circ \ldots \circ s_k] \circ \ldots \circ [s_1 \circ s_2 \circ \ldots \circ s_k]$ (brackets added for readability). The length of the repetition can vary and will be stated whenever this input is used. The string is constructed so that there are no repetitions within the repetition itself, i.e. the string that is repeated is heterogeneous. This means that the alphabet size is $k$.

**MaximumS:** A string where as many characters as possible are S-type. This is created by having each character in the string be 1-3 greater than the preceding character. The alphabet size is linear in the length of the string.

**MaximumL:** A string where as many characters as possible are L-type. This is created by having each character in the string be 1-3 smaller than the proceeding character. The alphabet size is linear in the length of the string.

**MaximumLMS:** A string where there are as many LMS-suffixes as possible. This is created by having every second character come from the “top half” of the alphabet, while the remaining characters come from the “bottom half” of the alphabet. The alphabet is set to have a fixed size of 256.
Since the running time of all algorithms but the naive is affected by the alphabet size*, it can be hard to compare the running time of input having a constant alphabet size with the running time of input having a linear alphabet size. This comparison, however, is necessary to compare the running time of the above inputs, and we will do our best to take this caveat into account.

11.1.1 A note on the number of LMS-suffixes in random data

We generated random and heterogeneous input of length 1,000,000 a thousand times and counted the number of LMS-characters. We found that for Random, 29.17% of all characters were LMS-characters with a standard deviation of $2.39 \cdot 10^{-4}$. For heterogeneous input, 33.33% of all characters were LMS-characters with a standard deviation of $2.04 \cdot 10^{-4}$.

This means that MaximumLMS is expected to have 50% LMS-suffixes, heterogeneous input is expected to have 33.33% LMS-suffixes and random input is expected to have 29.17% LMS-suffixes.

11.2 Experiments on Individual Algorithms

In this section, each algorithm is run on different input of equal length, to see the running time relative to the type of input. We then discuss the results, in particular any anomalies and unexpected behaviour.

11.2.1 Naive

The running time of the naive algorithm depends greatly on the implementation of the sorting algorithm being used. Java’s array sorting procedure is a stable, adaptive, iterative mergesort [13] that builds on TimSort [18], which itself uses ideas from [19]. The procedure has a worst-case time complexity of $O(n \log n)$, but is much faster when sorting partially sorted arrays, and arrays where part of the array is in ascending or descending order. The details of how these improvements work will not be investigated further.

As can be seen from Figure 11, the repetitive input is by far the slowest, with a repetition length of 20 being slightly slower than a repetition length of 100. The reason that these are slower than the rest, is likely due to the fact that all suffixes share a long prefix with other suffixes – for instance, if the repetition length is 20, $S[i...j] = S[i+20...j+20]$ for any $i, j$, where $j + 20 < |S|$. This means that to decide the order between $S_i$ and $S_{i+20}$, the algorithm has to compare

*The time for Counting-sort for the Skew algorithm and for determining the boundaries of suffix array buckets all depend on the alphabet size.
11.2 Experiments on Individual Algorithms

Figure 11: Running time of naive algorithm on different inputs. Note that MaximumLMS, MaximumL, MaximumS, Hetero and Random lie almost on top of one another.

![Figure 11: Running time of naive algorithm on different inputs. Note that MaximumLMS, MaximumL, MaximumS, Hetero and Random lie almost on top of one another.](image1)

Figure 12: Running time of naive algorithm on different inputs, divided by $n^2 \cdot \log n$.

![Figure 12: Running time of naive algorithm on different inputs, divided by $n^2 \cdot \log n$.](image2)

$|S_{1+20}|$ characters. The reason that a repetition length of 100 is faster than a repetition length of 20 is due to the fact that fewer suffixes share the same prefix, i.e. if the repetition length is 20, every 20th
suffix share the same prefix, while only every 100th suffix share the same prefix, if the repetition length is 100. This essentially means that the lower the repetition length, the slower the algorithm is expected to run.

Homogeneous input is faster than repetitive input, even though it can be seen as repetitive input, with a repetition length of 1. The reason for this speed up is likely due to the fact that the arrays that are compared are already in almost sorted order. That is, \( S_i \) and \( S_j \) are sorted, except for the sentinel, and it is always the case that \( S_i > S_j \), when \( i < j \), meaning that Java’s array sorting algorithm can utilize its speed ups mentioned above.

The rest of the inputs are much faster. For MaximumL, MaximumS and heterogeneous, it is the case that all characters are unique, meaning that only a single character comparison will be needed to determine the order of two suffixes. For MaximumLMS and random, it is the case that the chance of two suffixes sharing a long prefix is small, since the input consists of random characters, meaning that the expected number of characters that needs to be compared to determine the order of two suffixes will likely be small.

From Figure 12, it can be seen that the running time of both of the repetitive inputs and the homogeneous input seem to go towards a non-zero constant, when divided by \( n^2 \log n \), indicating that \( \Theta(n^2 \log n) \) is in fact their time complexity. For the rest of the input, the graph offers less information, but from the raw data, it seems that they go towards 0, indicating a time complexity where \( O(n^2 \log n) \) is not a tight upper bound. This is to be expected, as far fewer than \( n \) comparisons are needed to determine the order of two suffixes, as described above.

11.2.2 KA

From Figure 13, it can be seen that KA has the slowest running time when the input is heterogeneous, which is slightly slower than MaximumLMS. MaximumLMS will have a 50%/50% distribution of S- and L-type suffixes, while heterogeneous is expected to have the same. This means that KA does not benefit from choosing the smaller of the two sets, i.e. the amount of suffixes that need to be sorted in sample sorting is as large as possible. The reason that heterogeneous is slower than MaximumLMS, is likely due to the fact that its alphabet is linear in the input size, while it is constant for MaximumLMS.

Like heterogeneous input, random input will likely have roughly as many S- as L-type characters, though with greater variance than for MaximumLMS. Furthermore, random has a very small alphabet size, resulting in a faster running time than MaximumLMS and heterogeneous input.
The fact that the alphabet size of Repetitive100 strings is five times that of Repetitive20 strings, is likely the reason for the increased running time of Repetitive100.

Finally, MaximumS, MaximumL and homogeneous input is much faster than the others. Since homogeneous and MaximumL contain
only one S-type character (the sentinel) and MaximumS contains only one L-type character (the one preceding the sentinel), the sample sorting for these inputs consists of sorting just a single suffix, meaning that the sample sorting step is practically skipped, causing the speed up. The reason that homogeneous input is faster than MaximumL and MaximumS is likely due to the fact that it has a constant alphabet size, whereas the alphabet for MaximumL and MaximumS is linear in the length of the string.

Figure 14 shows that on all input, the running time seems to converge, albeit slowly for some input types, towards some non-zero constant when divided by $n$, indicating that the running time of the algorithm is $\Theta(n)$, as expected.

11.2.3 Skew

The more heterogeneous and the less repetitive the input is, the lower the probability of two 3-grams getting assigned the same rank is, and thus the faster Skew would be expected to perform. This means that, in expectation, Skew should perform well on MaximumLMS, MaximumL, MaximumS, heterogeneous and random input, while it should perform poorly on repetitive and homogeneous input. However, as can be seen from Figure 15, this is clearly not the case.

Skew has the largest running time when the input consists of as many LMS-substrings as possible, followed by heterogeneous, repetitive100 and random. The fastest input is MaximumL and MaximumS,
followed by homogeneous and repetitive20. MaximumL and MaximumS were fast and Repetitive100 was slow, as expected – but besides these, the running time of the different types did not follow the expectations.

The increased alphabet size of Repetitive100 relative to Repetitive20 is likely what is causing the difference in running time between the two.

Overall, Skew did not behave as we would have expected. We have no solid proof or reasoning for why Skew behaves as it does, but it seems that counting-sort is the main part of the algorithm which is significantly slower on MaximumLMS, heterogeneous, repetitive100 and random input than on MaximumL, MaximumS, homogeneous and Repetitive20 input. Regardless of input type, the theoretical running time of counting-sort is $O(n + k)$, where $n$ is the number of elements in the input array, and $k$ is the range of the input. This, however, was not what we observed in practice. MaximumL with large values of $n$ and $k$ had a much faster execution time for counting-sort, than MaximumLMS with small values of $n$ and $k$, indicating that, in fact, the input type matters quite a lot.

Figure 16 shows that on all input, the running time seems to converge towards some non-zero constant when divided by $n$, indicating that the running time of the algorithm is $\Theta(n)$, as expected.
11.2.4 SA-IS

Figure 17: Running time of SA-IS algorithm on different inputs. Note that MaximumL, MaximumS and Repetitive20 practically lie on top of one another.

Figure 18: Running time of SA-IS algorithm on different inputs, divided by $n$. 
From Figure 17, it can be seen that the slowest input to SA-IS is the heterogeneous input, followed by MaximumLMS. MaximumLMS contains more LMS-suffixes than heterogeneous input and as such, the sample that needs sorting is greater for MaximumLMS than for heterogeneous. Despite this, heterogeneous is slower. This is likely due to the fact that the alphabet size of MaximumLMS is constant, while it is linear in the length of the input for heterogeneous.

Following MaximumLMS is random. The main reason that random is faster, is likely that random contains fewer LMS-suffixes than both MaximumLMS and heterogeneous input, as described in Section 11.1.1. Furthermore, random has a constant sized alphabet that is a factor 64 smaller than that of MaximumLMS.

Repetitive input is expected to have the same number of LMS-suffixes, regardless of the repetition length. However, the alphabet size is five times larger when the repetition length is 100 than when it 20. This might explain why Repetitive100 is noticeably slower than Repetitive20.

For both MaximumL, MaximumS and homogeneous input, it is the case that there is only one LMS-suffix, namely the sentinel. This means that for all three inputs, the sample that needs sorting is very small, and so the sample sorting step is practically skipped, yielding a much faster running time than for other input. Homogeneous is faster than the others, which is likely due to its fixed alphabet size.

Figure 18 shows that on all input, the running time seems to converge towards some non-zero constant when divided by $n$, indicating that the running time of the algorithm is $\Theta(n)$, as expected.

11.2.5 MP

The MP algorithm was not tested on repetitive input, as the running time was too slow to achieve any meaningful results. This is due to the fact that the repetition heuristic has not been implemented, as described in Chapter 9.

Figure 19 shows that MP is slowest on heterogeneous, random and MaximumLMS input. Common for all three types of input, is that they have a relatively large amount of LMS-suffixes, as described in Section 11.1.1, meaning that MP has a large sample that needs sorting with MKQS. Heterogeneous is slower than MaximumLMS, despite having fewer LMS-suffixes. This can likely be attributed to the fact that the alphabet size is linear in the input length for heterogeneous input, while it is constant for MaximumLMS.

Random and MaximumLMS have roughly the same running time, despite MaximumLMS having more LMS-suffixes than random. This is probably due to random having more repetitions than MaximumLMS – since its alphabet size is much smaller than MaximumLMS,
Figure 19: Running time of MP algorithm on different input. Note that MaximumLMS and Random practically lie on top of one another, as does MaximumL and MaximumS.

Figure 20: Running time of MP algorithm on different types of input, divided by $n$.

and since two consecutive characters can be the same, there is a much greater chance of repetitions throughout the string.
Finally, MP is much faster when the input is MaximumS, MaximumL or homogeneous. This is due to the fact that since MaximumL, MaximumS and homogeneous input all have just one LMS-suffix, namely the sentinel, the sample that needs sorting has size one. This means that the sample sorting step can practically be skipped. The reason that homogeneous input is faster than MaximumL and MaximumS is likely due to it having a constant, very small alphabet size, whereas MaximumL and MaximumS both have linear alphabet sizes.

Figure 20 shows that on homogeneous, MaximumL and MaximumS input, the running time seems to converge towards some non-zero constant, when divided by $n$, indicating that the running time of the algorithm on this type of input is $\Theta(n)$. This makes sense since MKQS only sorts one LMS-suffix in each case and so we achieve the best case asymptotic time complexity of $\Theta(n)$ previously described in Section 9.2. On MaximumLMS and random input, on the other hand, the running time does not seem to converge towards a constant when divided by $n$, indicating that the running time on this type of input might be super-linear. In fact, since the expected linear number of sample suffixes of these input types are expected to have short common prefixes, one would expect a running time of $\Theta(n \log n)$ from MKQS which then dominates the total running time as explained in Section 9.2.
11.3 Experiments Comparing All Algorithms

In this section, all algorithms will be compared on the same input. The purpose of this is to compare the algorithms and investigate how well they perform relative to each other on various types of input.

As with the experiments on the individual algorithms, we discuss the results of the experiments, paying particular attention to unexpected results and anomalies.

11.3.1 Heterogeneous

On heterogeneous input, KA is expected to perform quite poorly, since the number of S-type suffixes is expected to be roughly the same as the number of L-type suffixes, resulting in a large sample that needs sorting. Similarly, there is likely a large number of LMS-substrings, meaning that the sample that SA-IS and MP need to sort is also large. On the other hand, Skew is expected to be quite fast, since every 3-gram will get a unique rank, meaning that there are no recursive calls. Finally, since no two suffixes start with the same character, the naive algorithm is expected to have a good running time as well, as the order of two suffixes can be determined after only one character comparison.

From Figure 21, it can be seen that KA is the slowest, followed by SA-IS, which is to be expected, as per the description above. Both the naive algorithm and Skew are expected to perform quite well on
heterogeneous input, and the naive algorithm is faster than both KA and SA-IS, while being slower than Skew. Finally, MP is the fastest, despite having to sort a sample of the same size as SA-IS.

11.3.2 Homogeneous

On homogeneous input, the naive algorithm and Skew are expected to perform quite poorly. For the naive algorithm, determining the order of two suffixes takes time linear in the length of the shortest suffix. For Skew, all 3-grams will get the same rank, except for the ones containing a sentinel, resulting in a maximum number of recursions. On the other hand, both KA, SA-IS and MP are expected to be relatively fast, since the number of S-type suffixes, and the number of LMS-suffixes, is just one, making sample sorting very fast.

![Figure 22: Running time of all algorithms on homogeneous input](image)

In Figure 22, it can be seen that the naive algorithm has been left out. For a homogeneous input string of length 30,000, the naive algorithm uses roughly 2.8 seconds on our test machine, which makes it practically impossible to plot alongside the other algorithms, in any meaningful way.

Besides the naive algorithm, Skew is by far the slowest, which is to be expected, as described above. KA is slightly faster than SA-IS, though their running times are pretty close to each other, which would be expected, since their induced sorting steps are almost identical. SA-IS is slightly slower, which is probably due to the fact that it scans the input three times during induced sorting – first scan places
LMS-suffixes, second scan places L-suffixes and the third scan places S-suffixes – while KA only does two scans. Finally, MP is the fastest, outperforming both KA and SA-IS. Although MP sorts the same sample suffix as SA-IS, its better performance might be attributed to the fact that MKQS terminates almost immediately when sorting only one suffix, while SA-IS still has to go through three scans of the input even when sorting only one suffix, i.e. SA-IS has a larger overhead.

11.3.3 MaximumL

On input consisting of as many L-type characters as possible, all algorithms are expected to perform well. There is only a single S-type character and only a single LMS-substring, namely the sentinel, which means the sample sorting of both KA, SA-IS and MP should be very fast. Furthermore, the input consists of all different characters, meaning that the number of comparisons needed for the naive algorithm is just one, and all 3-grams will be assigned a unique rank in Skew, making both algorithms have a fast running time.

Figure 23: Running time of all algorithms on MaximumL input

Figure 23 shows that SA-IS and KA are the slowest, but have similar running times. Since they practically have no sample sorting, this similarity in running time is to be expected, with SA-IS being slightly slower than KA, for the same reasons as described for homogeneous input. Skew and MP both outperform KA and SA-IS, with MP being slightly faster than Skew. As we saw when testing Skew in isolation, MaximumL and MaximumS were after all its best case inputs. Finally,
11.3 Experiments Comparing all Algorithms

The naive algorithm is the fastest. The reason for this is likely that the string, and thus all of its suffixes, is already sorted, making the array sorting much faster, as described in Section 11.2.1.

11.3.4 MaximumS

Since input consisting of as many S-type characters as possible (MaximumS) is practically equal to input consisting of as many L-type characters as possible (MaximumL), but reversed, MaximumS has the same characteristics when it comes to running time, as MaximumL. For this reason, the running times are expected to be similar, which is also the case, as can be seen in Figure 24.

![Figure 24: Running time of all algorithms on MaximumS input](image)

11.3.5 MaximumLMS

For input consisting of alternating S- and L-type characters, resulting in a maximum number of LMS-suffixes, KA, SA-IS and MP are expected to be relatively slow, since all three have a maximum number of sample suffixes that need sorting. For SA-IS and MP, this is trivially true, as their samples consist of all LMS-suffixes, while for KA, it is true since there are as many S- as L-type characters, resulting in the sample that needs sorting consisting of half of all suffixes. Skew and the naive algorithm, on the other hand, are expected to be relatively fast, as two suffixes likely have no common prefix, or a very short shared prefix. This means that few 3-grams will share the same rank.
in Skew, and the naive likely needs very few comparisons to distinguish two suffixes – for almost all suffixes, a single comparison will suffice.

As can be seen from Figure 25, KA is the slowest, followed by the naive. This indicates that indeed the 50/50 distribution of S- and L-types has a large impact on the performance of KA. Following the naive is SA-IS which is noticeably faster than KA, showing that even though the input should make both algorithms have a large running time, SA-IS outperforms KA. SA-IS is on par with Skew, despite Skew having low recursion depth. This reflects the relatively poor and somewhat unexpected behaviour of Skew on MaximumLMS input we saw earlier. Lastly, MP outperforms all the other algorithms, despite having to sort as large a sample as possible. However, while MP’s sample is large, the LMS-suffixes are not likely to share long prefixes, meaning MKQS will likely get close to its optimal, logarithmic recursion depth.

11.3.6 Random

On random input, the naive algorithm is expected to perform relatively well, as the expected number of characters that need to be compared to determine the order of two suffixes is low. Since the size of the alphabet is 4, there are $4^3 = 64$ different combinations of three characters, and thus Skew will likely divide the input into 64 different ranks – better than what would be expected for more ho-

Figure 25: Running time of all algorithms on MaximumLMS input. Note that Skew and SA-IS practically lie on top of one another.
mogeneous input, but worse than what would be expected for more heterogeneous input. KA is expected to be relatively slow, as there is expected to be roughly as many S- as L-type suffixes. Finally, as described in Section 11.1.1, the expected number of LMS-suffixes is approximately $n \cdot 0.29$, meaning that the number of sample suffixes for MP and SA-IS is lower than that of KA.

![Graph showing running time of all algorithms on random input](image)

Figure 26: Running time of all algorithms on random input

Figure 26 shows that the naive algorithm is the slowest. To keep the graph readable, the running time of the naive algorithm is not shown for input sizes above 13,000,000 characters. One factor in why the naive algorithm performs relatively worse compared to Maximum-LMS input could be that every other suffix in MaximumLMS strings are guaranteed to differ on their first character. Further, because the alphabet size is only 4, the length of common prefixes of suffixes is expected to be longer. Faster than the naive is KA, followed by Skew. Despite an expectation of few 3-grams having the same rank, Skew performs relatively poorly which was also observed when testing Skew in isolation. Faster than Skew is SA-IS, followed by MP which follows the expectation.

11.3.7 Repetitive100

For input containing a lot of repetitions, MP is expected to perform very poorly, as the repetition heuristic has not been implemented, as mentioned in Section 9.4.2. Furthermore, the naive algorithm is expected to perform poorly as well, as all suffixes will share a long
prefix with all other suffixes, meaning that a lot of comparisons have to be made, to distinguish two suffixes. For the same reason, Skew is expected to be relatively slow, as a lot of 3-grams are expected to have the same rank. Similarly, KA and SA-IS have to sort a lot of equal substrings, which should make for a relatively slow running time, due to a large recursion depth.

![Figure 27: Running time of KA, SA-IS and Skew on repetitive input with a repetition length of 100](image)

In Figure 27, the naive algorithm and MP have been left out, as their running times are too great to produce a meaningful plot, where they are compared alongside the other algorithms.

Figure 27 shows that Skew is the slowest, followed by KA and SA-IS, as expected. There is quite a gap between KA and SA-IS, which likely comes from SA-IS having a smaller sample to sort than KA.

Similar results were seen for input with different repetition length. In general, it would be expected that the shorter the repetition, the longer the shared prefixes will be, and thus resulting in a larger running time. However, the fact that the alphabet size is linear in the repetition length seems to outweigh this, and we generally observed larger running times for larger repetition lengths.
REAL WORLD DATA

The data from Manzini and Rastero [20]* seems to be the de facto standard used when benchmarking SACAs with real world data, and is used in both [3], [4] and [10]. All of the data consists of strings made from the alphabet \{a, c, g, t\}. We have computed the longest common prefix (LCP) and average match length (AML) of each input, which can be seen in Table 1, along with the length of each input. The LCP column denotes the longest common prefix among all pairs of suffixes of the input. The AML column denotes the average match length, as defined by Sadakane [21], for each input, and is a measure of how many comparisons are needed, on average, to distinguish two suffixes following each other in the suffix array of the input. The last five columns in the table denote the running time in milliseconds for each algorithm. For each row, the fastest algorithm is marked in green, the second fastest in yellow, the third fastest in orange and the fourth fastest in red.

As can be seen in Table 1, the general trend is that SA-IS is faster than MP on smaller input, while MP is faster on larger input. Both SA-IS and MP are faster than Skew and KA, while Skew is faster than KA on small input, but slower on large input. Finally, on all input, the slowest algorithm is the naive.

The running time of the naive algorithm depends greatly on the average number of characters that need to be compared, to distinguish two suffixes. The AML is a good measure for this, and it can be seen that the greater the AML, the greater the running time of the naive algorithm. For instance, hsy and at3 have roughly the same size, but the naive algorithm is much faster on the larger of the two, which has a much lower AML. The same can be seen for mmx and at4.

The fact that KA outperforms Skew on the larger data sets, but is outperformed on smaller data sets, indicates that KA has a larger constant overhead than Skew.

For KA and SA-IS, the LCP and AML does not directly influence running time, and as such, there is no noticeable gap between inputs of similar lengths but with differing LCP and AML.

LCP and AML are not measures of how repetitive a string is. However, if a string has a large LCP and/or AML, it likely has some repetition, i.e. if a string has a large AML, a lot of suffixes likely share a long common prefix, meaning that the prefix repeats throughout the string. Since the repetition heuristic for MP has not been implemented, MP is very sensitive to repetitions. This is likely the reason

---

*Available at http://people.unipmn.it/~manzini/dnacorpus/
### Table 1: Datasets and their length, LCP and AML. Values in algorithm columns are running time in milliseconds. The fastest algorithm for each input is marked in green, the second fastest in yellow, the third fastest in orange and the fourth fastest in red.
that MP is faster on \textit{at}_4 than on \textit{mmx}, despite \textit{at}_4 being the largest and faster on \textit{at}_3 than on \textit{hsy}, despite \textit{at}_3 being the largest – the greater LCP and AML of the \textit{mmx} and \textit{hsy} indicates that these strings might be more repetitive than \textit{at}_4 and \textit{at}_3.
For all algorithms, except the naive, the slowest inputs were Heterogeneous and MaximumLMS. For KA, SA-IS and MP, this is to be expected, as both types of input have a lot of LMS-suffixes, and is expected to have an equal amount of S- and L-type suffixes. Despite Heterogeneous having fewer LMS-suffixes than MaximumLMS, this was the slowest input for both SA-IS and MP, which can likely be attributed to Heterogeneous having an alphabet size that is linear in the input length, while MaximumLMS has a fixed and relative small alphabet size of 256. Finally, Skew would be expected to be fast on both types of input, but as described in Section 11.2.3, this was not the case.

The fastest inputs for all but the naive algorithm is Homogeneous, MaximumL and MaximumS. For KA, SA-IS and MP, this is to be expected, since the sample that needs sorting consists of just a single suffix. For Skew, MaximumL and MaximumS are fast as expected, but Homogeneous would be expected to be slow, as all 3-grams will be assigned the same rank. However, as described in Section 11.2.3, this is not the case.

The general trend for the experiments conducted on the artificial data is that MP outperforms the other algorithms, except when faced with repetitive input, or when the naive can utilize its speed ups. This is even the case for input where MP is expected to be relatively slow, such as MaximumLMS.

On repetitive input, SA-IS is clearly the fastest. This is to be expected, since SA-IS outperforms KA on all inputs where the sample that needs sorting is not too small, and Skew is expected to have a relatively large running time, since repetitions will mean that a lot of 3-grams are assigned the same rank, resulting in a large number of recursions.

The same trend is seen on the real world data, where MP outperforms all other algorithms, when the input is large enough, while SA-IS is faster on smaller input. The fact that SA-IS is faster on smaller input, indicates that MP has a larger constant overhead than SA-IS.

All in all, on input that is not too repetitive, and where the naive algorithm cannot use its speed ups, the trend is that for large enough input, MP outperforms SA-IS, which outperforms KA, which outperforms Skew, which then again outperforms the naive algorithm. This relative order is also the order found by Puglisi, Smyth, and Turpin [10], of course excluding SA-IS, which was published later than [10]. Thus we have replicated their results.
SA-IS was probably the easiest algorithm to implement, followed by MP, without the repetition heuristic. Skew was also pretty straightforward, while KA was immensely harder to implement, primarily because it at first wasn’t clear to us why the algorithm worked. Taking performance relative to implementation difficulty into consideration, SA-IS might be the algorithm of choice, as it is easy to implement, asymptotically optimal and is only outperformed by MP, when the input is not too repetitive. However, on real world data, MP might be better, as it is also fairly easy to implement without the repetition heuristic, and outperforms all other algorithms for large inputs.
14 CONCLUSION AND FUTURE WORK

14.1 CONCLUSION

The aim of this thesis has been to describe, examine and discuss a number of different suffix array construction algorithms, with particular focus on algorithms that have linear time complexity in the worst case. The algorithms have been described using one common notation, and an effort has been put into clarifying and elaborating on parts of the original papers that we found to be insufficiently detailed, including proofs of why they work. The use of a common notation makes it much easier to understand how one algorithm relates to another, and serves as a small descriptive framework of suffix array construction algorithms.

Five algorithms have been implemented. One is the most naive way to construct suffix arrays, three algorithms, KA, SA-IS and Skew, are worst-case linear time algorithms and finally, the last algorithm, MP, aims to be fast in practice, while it is not guaranteed to have linear time complexity in the worst-case. Based on our implementations of these algorithms, we have evaluated their performance both individually and relative to one another.

The descriptions of the algorithms, and how easy they were to implement based on these descriptions, varied quite a bit. SA-IS and MP in its simplest form were relatively easy to implement. However, understanding why they actually construct the correct suffix array of a given input, took some effort. Skew was also relatively easy to implement, and the reasoning for why it works seems simpler, than that of SA-IS and MP. Finally, the description of KA left out a lot of details that we have had to figure out ourselves. Both the understanding of why KA works and actually implementing it required a lot of effort and several attempts, making it, by far, the most cumbersome algorithm to implement and reason about.

A number of experiments have been conducted, investigating the behaviour of each algorithm individually, run on different types of input, as well as comparing the algorithms against one another. These experiments can be used as a frame of reference for the performance of the different algorithms on different input, and the tendencies will likely generalize to data not explored in this thesis.

From the experiments, it was found that generally, MP outperforms SA-IS, which outperforms KA, which outperforms Skew. This means that the algorithms that were the simplest to implement, were also the ones generally performing the best.
Despite our implementation of MP being the simplest possible, without the repetition heuristic, MP was faster than all other algorithms on data from the real world, for sufficiently large input. This indicates that when faced with real world data, algorithms such as KA, SA-IS and Skew, that guarantee an optimal worst-case asymptotic running time, still have a ways to go, before being able to compete with algorithms such as MP that are made to be fast in practice, instead of guaranteeing an optimal worst-case asymptotic running time.

14.2 Future Work

Despite not implementing the repetition heuristic for MP, it generally performed very well in comparison with the other algorithms. An obvious candidate for future work would be to examine, describe and implement this repetition heuristic in order to study the benefits and caveats it will have on the performance of MP.

Another candidate for future work would be to examine absolute running times instead of relative running times. In the experimental part of this thesis, only the order of algorithms on different input and the order of input on different algorithms has been examined, while there has been paid no attention to the absolute running times of the algorithms. One reason to look at this might be to discuss what is feasible and what is not within a given time frame.

Finally, it might prove interesting to look into the space usage of different suffix array construction algorithms. While this has not been the focus of this thesis, it is of course an important aspect, when considering sufficiently large input.
BIBLIOGRAPHY


87


