EFFICIENT IMPLEMENTATIONS OF SUFFIX TREES

MARC SKODBORG, 201206073
SIMON FISCHER, 201206049

MASTER’S THESIS
June 2017
Advisor: Christian Nørgaard Storm Pedersen
ABSTRACT

This thesis intends to explore different ways of constructing suffix trees, focusing in particular on the algorithm by Farach [6] as it remains largely unexplored in practice.

As a frame of reference for the exploration of this particular algorithm, two other algorithms is presented, namely McCreight’s algorithm [16] along with a naive approach.

These algorithms have been implemented with the aim of exploring the effects observed on construction time in practice of various characteristics of data.

This exploration was conducted through the construction of various types of artificial data, followed by experiments with real data to aid in determining the degree of influence of these characteristics on practical examples.

Particularly when experimenting with various alphabet sizes, of which the performance of Farach’s algorithm is independent contrary to McCreight’s algorithm being dependent hereupon, it was shown to what degree this difference in dependence separates the two algorithms in practice.

When considered collectively, the conducted experiments will give an impression of performance in practice of each of the algorithms, along with our supplied implementations hereof, which will hopefully generalize to data types not featured in this thesis.
## CONTENTS

1 **INTRODUCTION** 1  
1.1 Motivation 1  
1.2 Contribution 2  
1.3 Division of labour 3

### I THEORETICAL FRAMEWORK 5

2 **PRELIMINARIES** 7  
2.1 Alphabet 7  
2.2 String 7  
2.3 Suffix and Prefix 7  
2.4 Trie 8  
2.5 Compacted Trie 8  
2.6 Suffix Tree 9  
2.7 LCP and LCA 10  
2.8 Suffix links 10  
2.8.1 LCP tree 11

### II NAIVE ALGORITHM 13

3.1 Time complexity analysis 14

### III MCCREIGHT’S ALGORITHM 15

4.1 Time complexity analysis 18

### IV EFFICIENT LOWEST COMMON ANCESTOR RETRIEVAL 21

5.1 Preprocessing 21  
5.2 LCA queries in constant time 27

### V FARACH’S ALGORITHM 29

6.1 Algorithm overview 29  
6.2 Constructing the odd suffix tree $T_{\text{odd}}$ 30  
6.3 Constructing the even suffix tree $T_{\text{even}}$ 32  
6.4 Overmerging of $T_{\text{odd}}$ and $T_{\text{even}}$ 35  
6.5 LCP tree construction 39  
6.6 Final adjustments 43  
6.7 Choices made different 46  
6.8 Time complexity analysis 49

### VI PRACTICAL EXPERIMENTS 51

7 **EXPERIMENTS** 53  
7.1 Environment 53  
7.2 Implementations and how to use 54  
7.3 Verification of implementation correctness 54  
7.4 Data choices 56  
7.4.1 Artificial data 56  
7.4.2 Real data 57
8 Artificial data 59
  8.1 Initial impression 59
  8.2 All identical characters 60
  8.3 All different characters 63
  8.4 Periodic strings 65
  8.5 Varying alphabet sizes 71
    8.5.1 Fixed alphabet sizes 71
    8.5.2 Relative alphabet sizes 77
9 Real data 83
  9.1 DNA 83
  9.2 English text 85
  9.3 Summary 87
10 Discussion 91
11 Conclusion 95
  11.1 Future work 96
INTRODUCTION

This thesis will engage in different ways of constructing suffix trees, focusing on the algorithm by Farach as described in [6] with reflections on this algorithm being made in relation to the naive approach to suffix tree construction, along with the algorithm presented by McCreight in [16].

In this introduction, there will be a brief motivation for constructing said suffix trees at all. Following this will be a description of our contribution to the area of constructing these, immediately followed by an overview of the content of the thesis at hand. Lastly, the division of labour will be shortly outlined.

1.1 MOTIVATION

In many of today’s fields of work, queries on datasets consisting of large strings are common. An example of a such field is biology, where the growth of biological data available at GenBank is exponential, roughly doubling the number of available bases every 18 months since 1982[9, 10]. Working with datasets of these magnitudes requires indexing data structures, with the suffix tree data structure allowing for efficient searching for both exact patterns and different regularities, such as palindromes[11] and tandem repeats[12], in the data at hand.

In brief, a suffix tree is a compacted trie of all the suffixes of a given string. To ensure no suffix being a prefix of another, a unique symbol is often appended to the end of the given string. This symbol, usually referred to by $, will ensure a one-to-one correspondence between suffixes of the string and leaf nodes in the suffix tree. One such tree is illustrated in Figure 1.

This data structure, while obviously taking time to construct initially, will allow for greatly improved performance on all subsequent queries. These queries answers problems such as determining the longest repeated substring, the longest common substring and pattern matching [1], along with finding tandem repeats [12] and more.

Algorithms constructing the suffix trees themselves are subject to a variety of strategies. And, as a consequence of these construction strategies, not all algorithms perform equally well on all types of datasets. Particular characteristics present in the data at hand might influence the performance of these algorithms. It may even do so to an extent that makes the choice of which algorithm to use not being without importance.
1.2 CONTRIBUTION

While having a well-documented theoretical worst-case complexity, Farach’s algorithm [6] remains largely unexplored in practice [25, p.136]. Being a linear-time complexity algorithm, its claimed advantage over other such characterized algorithms lies in its independence of the size of the alphabet, whereas McCreight’s algorithm [16], being such another linear-time complexity algorithm, claims only to run in linear-time under the assumption that the alphabet is of constant size. In spite of it not being a linear-time complexity algorithm, the naive approach to constructing suffix trees, outlined in Chapter 3, is both simple and broadly well-understood and thus relatable, letting for a suitable baseline algorithm for comparison of the aforementioned two algorithms of Farach and McCreight, respectively.

Our contribution will be an account of the theory behind as well as of each algorithm, backed by a working and available implementation of each of the three.

Further, as both Farach’s and McCreight’s algorithms are of linear-time complexity in theory, we will seek to understand and document the differences in performance hereof in practice, and will do so for various types of datasets. These various types of datasets will be both artificially constructed ones, intended to reflect certain aspects of data potentially influencing construction times, and real data, intended to indicate to which degree these aspects of data are present in potential real use cases. While not comprehensive, we hope the chosen experiments will form a succinct impression of the relative performances of the algorithms, generalizing to types of data not featured in this thesis.

In particular, we are interested in to what extent the advantages of Farach’s algorithm [6] being independent of the size of the alphabet shows in practice, as opposed to McCreight’s algorithm [16].
Thesis structure

The thesis is split up into two separate parts, one concerning each of theory and practice.

The part concerned with theory will have preliminary theory covered initially in Chapter 2, allowing for delving into each of the algorithms in order of increasing theoretical involvement. Chapters 3 and 4 will then cover the naive approach to suffix tree construction and McCreight’s algorithm, respectively.

Following these will be Chapter 5, outlining theory utilized by Farach in his algorithm, which will then be covered in Chapter 6 itself. Each of these chapters will cover the theoretical time complexities of their respective algorithms as well.

In the next part, concerned with the relative performances of the three suffix tree construction algorithms, Chapter 7 will introduce and outline in more details the structure of the process of experimentation as undergone. These experiments have been split into two general categories, namely artificial data types covered in Chapter 8 and real data types, covered in Chapter 9.

Chapter 10 will gather remaining points not outlined throughout the discussion of the experiments themselves, with Chapter 11 concluding on the work in this thesis, along with pointing out envisioned future work.

1.3 Division of Labour

Chapter 5 was made exclusively by Simon, whereas the rest of this thesis is the work of both of us.
Part I

THEORETICAL FRAMEWORK
In this section, the basic concepts and notation used throughout will be declared, based on the definitions of these as found in [25]

2.1 ALPHABET

An alphabet, denoted $\Sigma$, is a non-empty set of symbols. The alphabet is usually considered being of finite size, but this is not always the case, as will be evident later. All such alphabets can be converted to an integer alphabet as well, so assuming integer alphabets, as we will onwards, is not a limitation. One might even argue that only integer alphabets are possible in the world of computing, due to the binary representation of symbols. Characters and symbols will be used interchangeably throughout to reference elements of an alphabet.

2.2 STRING

A string $S$ is a finite sequence of symbols from an alphabet, such that $S \in \Sigma^n, n \in \mathbb{N}$. The empty string is denoted with $\lambda$. The length $n$ of a string $S$ is its cardinality, $|S| = n$. If nothing else is specified, $n$ will be used throughout to refer to the length of a given string. The symbol found at an index, $i$, in the string is written as $S[i]$. A substring is defined by a range of positions and will be written as $S[i..j]$ for $i, j \in 1..n, i \leq j$.

2.3 SUFFIX AND PREFIX

A suffix of a string $S$ is a substring $S[i..n]$ for $i \in 1..n$. For at given $i$, the $i$'th suffix will reference the string $S[i..n]$. A suffix is a proper suffix if $i > 1$, such that $|S[i..n]| < n$ and the string is not a suffix of itself.

Likewise, a prefix of a string $S$ is a substring $S[1..j]$ for $j \in 1..n$, referenced as the $j$'th prefix. A prefix is as proper prefix if $j < n$ such that again, $|S[1..j]| < n$.

Due to the identity $x = \lambda \lambda$, the empty string $\lambda$ is considered both a proper suffix and a proper prefix.
Let $X$ be a set, $X = \{x_1, x_2, \cdots, x_m\}$ of pairwise distinct strings over an alphabet $\Sigma$.

A trie, as seen in figure 2, is a tree with exactly $m$ leaves, one for each string in $X$. The edges between nodes in a trie have labels corresponding to the symbols in the strings contained in $X$, one character on each edge with no edge being empty. A path from the root node to some node $v$ forms a string $s$ when symbols visited along the path are concatenated. The node $v$ is said to represent the string $s$, with the root node representing the empty string. A node and the string it represents will be used interchangeably throughout. Particularly, if $v$ is a leaf node, then the path from the root to $v$ should spell out a string in $X$. All strings in $X$ should be spelled out by each their leaf node. All children of a node should be pairwise different which means that two children of a node cannot represent the same string. This effectively means that the downward edges leaving a node should all be labeled with pairwise different symbols.

A compacted trie, as seen in figure 3, is a trie as described above, but modified such that nodes with only one child is eliminated and the represented symbol added to the child node instead. This then leaves a tree in which edges are no longer limited to a single symbol, but instead contains strings. For a trie it is required that the symbols on the edges down to all children of a node are pairwise different. In the
2.6 Suffix Tree

A suffix tree is a compacted trie over the set of all suffixes of a given string, including the empty suffix $\lambda$ and the string itself. The leaf nodes in the tree are further annotated with each their id $i$, indicating that they represent the $i$'th suffix. To avoid having internal nodes represent suffixes, as would be the case if some suffix was a proper prefix of another suffix, a unique symbol $\$ \in \Sigma$, which is appended to the given string, $S$. This ensures that no suffix of $S$ is a proper prefix of another suffix, thus also ensuring that all suffixes are represented by leaf nodes, and none by internal nodes. For integer alphabets, this unique symbol could translate to $\$ = k + 1$, where $k$ is the highest integer found in the alphabet.

Suffix trees will have exactly $n + 1$ leaves, one per suffix and an extra for the appended unique symbol, and at most $n$ internal nodes, yielding a total of at most $2n + 1$ nodes and $2n$ edges. If the depth of the tree is considered 1 at the root node, and incremented by one each time a downward edge is followed, the suffix tree has a maximum possible depth of $n + 1$ with $n + 1$ leaf nodes.

A suffix tree is said to be sorted if the downward edges of all nodes are lexicographically sorted.

It is assumed that leaf nodes are accessible in constant time given their id, which is possible using typical table lookups on these id's.

When illustrating suffix trees throughout, they will be shown as having string labels on edges. This is only for pedagogical reasons, and these strings will not actually be held in memory. Allocating space for all $n$ suffixes of a string of length $n$ will require $O(n^2)$ space, as well as $O(n^2)$ time to copy all suffixes to their respective...
edges. Therefore, all internal nodes will contain only the length of the represented string, along with the id of a leaf node descending from this node. This is only $O(1)$ space, yet it is enough to retrieve the actual edge labels, should one wish to do so. No information is thus lost, but the space required for representing the suffix tree is reduced to $O(n)$ and no time is spent copying strings.

2.7 LCP and LCA

Given two strings, $x_i$ and $x_j$, their longest common prefix, in short LCP, is defined as the longest string $u$ that is a prefix of both $x_i$ and $x_j$. This will be written as

$$u = \text{LCP}(x_i, x_j)$$

Let $v$ denote the root node in the minimum size subtree that contains both nodes $v_i$ and $v_j$. The node $v$ is then the lowest common ancestor, in short LCA, for $v_i$ and $v_j$. This will be written as

$$v = \text{LCA}(v_i, v_j)$$

Remembering that a string and a node representing said string are considered the same thing, the above definitions gives the following property of a suffix tree:

$$\text{LCA}(v_i, v_j) = \text{LCP}(x_i, x_j) \quad i, j \in 1..n$$ \hfill (1)$$

Let $x_i$ and $x_j$ represent the $i$’th and the $j$’th suffix, respectively, and then let $v_i$ and $v_j$ be the nodes representing $x_i$ and $x_j$. As $v = \text{LCA}(v_i, v_j)$ is their LCA, i.e. the node furthest from the root node in the suffix tree that their two paths have in common, then this node must also represent the longest common prefix of the two suffixes, i.e.

$$u = \text{LCP}(x_i, x_j).$$

2.8 Suffix Links

A suffix link is a construction available on all internal nodes except for the root node. These internal nodes all represent some string, $a\alpha$, where $a \in \Sigma$ is a single character and $\alpha$ is a substring of $S$, possibly being empty. The suffix link of a such node $u$ is then a link pointing to the node in the tree representing the string $\alpha$. This node found by following the suffix link from node $u$ will hereafter be referred to as $s(u)$. The node $s(u)$ representing $\alpha$ must exist, as $\alpha$ is the prefix of some suffix and $a\alpha$ is represented by a node, indicating that at least two suffixes share this string. Consequently, the two following suffixes must share exactly $\alpha$. This can be seen in Figure 5 where the suffix link from the node representing string “ana” points to the node representing “na”
A relation between the depths of two nodes at both ends of a suffix link is as follows: By following a suffix link from node $u$ at depth $d(u)$ to node $s(u)$ in the suffix tree, this node $s(u)$ must be found at a depth of $d(s(u)) \geq d(u) - 1$, where the depth of a tree grows downwards from the root node. This is because each ancestor $v$ of $u$ different from $u$ itself and the root node will have a distinct longest prefix with $u$. Each of these ancestors will also, as argued previously, have a suffix link to some node $s(v)$. As these nodes $s(v)$ will all too have a distinct longest prefix with $s(u)$, they must all be on the path from the root node down towards $s(u)$, and thus be ancestors of $s(u)$. As there are $d(u)$ ancestors $v$ of $u$, and each, except for the root, has a corresponding distinct node $s(v)$, if no other nodes exist on the path, the minimum number of nodes visited on the path from the root node to $s(u)$ must be exactly the $d(u) - 1$ suffix link nodes from ancestors of $u$.[17, sec. 29.1]

### 2.8.1 LCP tree

Consider a suffix tree in which all suffix links has been created. These suffix links will form an overlaid tree structure different from that of the suffix tree itself. This tree will be named the LCP tree. An example of a suffix tree along with its suffix links forming the LCP tree is shown in Figure 5.

Recall that following a suffix link from an internal node will lead to another node representing a string of length one less. Therefore, the depth of a given node in the LCP tree, i.e. the number of suffix links to follow from that node to reach the root node, corresponds to the length of the string represented in this particular node.

Thus, as every internal node $v$ in a suffix tree is the LCA($l_i, l_j$) of at least one unique pair of leaf nodes $l_i$ and $l_j$, and as their LCP($l_i, l_j$) is represented by node $v$ as stated in (1), the length of the LCP($l_i, l_j$) can be found by consulting the LCP tree in which the depth of $v$ corresponds exactly to this length, due to the nature of suffix links.
As seen on Figure 5, suffixes 2 and 4, "anana$" and "ana$" respectively, share the proper prefix "ana" as represented by $v = \text{LCA}(2, 4)$. This node $v$ is located at a depth of 3 in the LCP tree, matching the length of "ana".
NAIVE ALGORITHM

Suffix trees can be constructed in a straightforward manner by processing each of the suffixes of $S$ one by one, using what will be referred to as the naive algorithm.

Essentially, this algorithm traverses the tree aiming to find the position in that current state of the tree at which to insert the suffix in question, make the necessary adjustments to the tree once this position has been determined, and continues to process the next suffix in the same way in this altered tree, until no more suffixes are left and all has been inserted.

Algorithm description

Consider string $S$ of length $n$ with suffixes $s_i, i = 1..n$. Processing the first suffix is easy: it is simply added as a leaf node descending directly from the root node. Each successive suffix from here onwards must be processed as follows: Let $T_i$ be the suffix tree as resulting from processing the $i$'th suffix. Traverse $T_i$ to determine the node $v$ representing the string with the longest possible prefix matching a prefix of $s_{i+1}$. This node $v$ may either exactly represent $\text{LCP}(v, s_{i+1})$, or it may represent a string longer than this. In case it represents exactly their common string, a new node $v'$ is simply inserted below $v$ to represent $s_{i+1}$. Otherwise, if the string of $v$ is longer than $\text{LCP}(v, s_{i+1})$, i.e. the position sought is to be found at the edge going into $v$, a new internal node $p$ is created. This node $p$ will be added as child to the parent of $v$, and will represent the string $\text{LCP}(v, s_{i+1})$, with $v$ itself being moved down to descend directly from $p$ along with $v'$. Once all suffixes has been processed in this way, the result will be a suffix tree of $S$. Example of the steps can be seen in Figure 7.

Figure 6: The four suffixes of $S = \text{aab}\$ are iteratively inserted into the tree, resulting in the suffix tree of $S$
14 Naive Algorithm

Figure 7: The two cases of insertion once node \( v \) has been located. In 1), the location was exactly \( v \), and \( v' \) is simply added as a child. In 2), internal node \( p \) is introduced, having length \( \text{LCP}(v, v') \) and having both \( v \) and \( v' \) as immediate descendants.

3.1 Time Complexity Analysis

Traversing the tree and determining the location of every node inserted by character comparison along edges, however, leads to a time complexity of \( \Theta(n^2) \) for the naive algorithm. Speedier alternatives to this construction algorithm exists, as the following two algorithms are examples of.
Like the naive algorithm, McCreight’s algorithm [16] for construction of suffix trees, hereinafter called McCreight’s, builds the suffix tree by iteratively processing a compacted trie of the strings $S[i..n]$ for $i = 1..n$, inserting the $i$’th suffix in iteration $i$. However, McCreight’s utilizes tricks to speed up the running time to $\Theta(n)$. The initial tree in each iteration $i$ of the algorithm will be referred to as $T_i$, whereas when the suffix $i + 1$ has been added, the resulting tree will be referred to as $T_{i+1}$. The following description will be based on the explanation of the algorithm found in [25, sec. 5.2.2] and [17, sec. 29.2.2]

**Algorithm description**

Specifically for this construction algorithm, the notion of a *head* and *tail* of a string is introduced. These are defined as follows [25, sec. 5.2.1]:

*head*($i$) is the longest LCP($S[i..n], S[j..n]$) for all $j < i$

i.e. the longest LCP of the $i$’th suffix and all previously considered suffixes.

*tail*($i$) is the string such that $S[i..n] = \text{head}(i)\text{tail}(i)$,

i.e. the string that, when concatenated with head($i$), will match the $i$’th suffix $S[i..n]$.

Initially, the tree $T_1$ contains the root node and one leaf node representing the 1st suffix descending directly from the root node. At iteration $i$ in the algorithm, suffix $i + 1$ is inserted into the tree $T_i$, yielding a new tree $T_{i+1}$. As suffix $i + 1$ is partitioned into head($i + 1$) and tail($i + 1$), each iteration tries to locate head($i + 1$), as tail($i + 1$) is trivial to insert afterwards; it always descends directly from the node representing head($i + 1$). This is repeated until all $n$ suffixes has been inserted, resulting in the full suffix tree $T_n$. This is assuming that the string $S$ has had the unique character $\$$ appended to the end, and the length $n$ is including this unique character.

**Speeding up scanning**

Searching for the location of head($i + 1$) for insertion of tail($i + 1$) to represent suffix $i + 1$ in the tree $T_i$ is done using two different strategies, depending on what is already known to be contained in $T_i$. If what is searched for is known to be located within the tree somewhere, it suffices to determine in which direction to search, i.e.
Figure 8: Slowscan and fastscan illustrated. Relation symbol $\equiv$ indicates an equivalence test being performed, whereas $\neq$ indicates that no test is performed since equality is already known, thus allowing for skipping of the comparison of characters. Note how only the first character of an edge is compared to a character in $S$ during a fastscan which edge to pass, and then skip the actual character-by-character comparisons knowing that they will all match anyway. This strategy will be referred to as fastscan. Otherwise, if it is not known whether what is being searched is within the tree or not already, a character-by-character search along the edges until a mismatch is spotted is necessary, as is performed in the naive algorithm. This procedure will be named slowscan.

The slowscan is a straightforward traversal, whereas the fastscan may be slightly less so. The fastscan is performed as follows: Starting in a node $u$ looking for pattern $v$, find the child node $c$ of $u$ for which the edge leading towards it starts with a character matching the initial character of $v$. Now, skip the prefix of $v$ of length same as the length of the label on the edge towards child $c$, and continue looking for the remainder of $v$ among the children of node $c$ until $v$ is covered entirely, completing the scan ending in either a node or at an edge. These two strategies for searching the tree for a string are illustrated in Figure 8

**Suffix link shortcuts**

Another speed-up relative to the naive algorithm, and the reason for splitting up the $i$'th suffix into head($i$) and tail($i$) is that it allows for creating potential shortcuts for later iterations through the use of suffix links. Consider the following situation: Suffix $i + 1$ must be inserted into $T_i$. This tree already contains a leaf node $l_i$ representing the $i$'th suffix. If there is an internal node $v$ representing some string
Figure 9: The leaf node \( i \) is the most recently inserted node in \( T_i \). As \( v \) is an internal node on the path from the root node to \( i \), another leaf node \( j \) with \( j < i \) will exist below \( v \). This means that node \( j + 1 \) must too exist somewhere in \( T_i \). As \( i \) and \( j \) shares a prefix \( a\alpha \) represented by \( v \), suffixes \( i + 1 \) and \( j + 1 \) must too share a prefix \( \alpha \). The suffix link from \( v \) will point directly to this node \( v' \) representing \( \alpha \), from which the leaf node \( i + 1 \) must descend

McCreight’s course of action

The way of operation of the algorithm is as follows: In the \( i \)’th iteration, suffix \( i + 1 \) must be inserted as a leaf node in \( T_i \). The iteration starts off in node \( v = \text{head}(i) \); this is the node directly below which leaf node \( i \) can be found. In case \( v \) is the root node, there is nothing else to do but slowscan from the root to find the position of suffix \( i + 1 \) and then insert it.

If \( v \) is not the root node, however, it must be an internal node representing some non-empty string \( a\alpha \). Obviously, \( v \) will not have a suffix link accessible at this point, as this would imply that the location of suffix \( i + 1 \) has already been determined in previous iterations, and it cannot be constructed until at the end of this \( i \)’th iteration when \( \text{head}(i + 1) \) has been located, to which the suffix link from \( \text{head}(i) \) should point. However, the parent of \( v \) might have one. If this is the case, this suffix link is followed, and the remainder of the \( i + 1 \)’th
Figure 10: An iteration of McCreight starts off in head(i). In 1), the parent node of head(i) is consulted in case a suffix link exists from here. If so, this is followed as shown in 2). Because this suffix link is followed, the label on the edge between head(i) and its parent must also be found below the end of the suffix link, so fastscan is utilized to determine s(head(i)) in 3). In 4), if the end of the fastscan is in a node, slowscan is used to find the remainder of the string head(i+1).

suffix is searched for from this point onwards using fastscan. Once the location of head(i+1) has been determined, a node representing it can be inserted if not already existing, and the remainder of suffix i+1, namely tail(i+1), is added as a leaf node below head(i+1). This procedure is sketched in Figure 10.

Once the suffix i+1 has been inserted, a suffix link can be created between head(i) and head(i+1) at this point. It will then be available as a shortcut later, just as the one utilized from the parent of head(i) and head(i+1) in this iteration was created in a previous iteration.

Lastly, head(i+1) and tail(i+1) should both be updated to prepare for the next iteration. Once all iterations have been completed, a suffix tree T of S should result.

4.1 Time Complexity Analysis

The algorithm iteratively inserts n leaf nodes, and apart from what happens during slowscan and fastscan, every other action can be implemented in time O(1), totaling Θ(n) besides the time spent scanning. If fastscan and slowscan are both within O(n) too, then McCreight can be concluded to perform within Θ(n).

Time complexity of Slowscan

[25, sec. 5.2.2] The algorithm determines where to insert head(i+1) based on the known position of the node s(head(i)), the node found at the end of the suffix link from head(i). Because the algorithm keeps track of and makes sure this node s(head(i)) is always available, slowscan can be guaranteed to perform no more than O(n) character comparisons. The reasoning behind this bound is based on the following observation, described in Lemma 4.1.1, cited from [25, p. 5.2.1]
Lemma 4.1.1 (5.2.1 in [25]) Let head(i) = x[i..i + h] for integers 1 ≤ i ≤ n and −1 ≤ h ≤ n − i. Then x[i + 1..i + h] is a prefix of head(i + 1)

This states that s(head(i)) is in fact a prefix of head(i + 1). Thus, the number of character-wise comparisons of these two, as performed by slowscan, is the length of the remainder of head(i + 1) once the prefix s(head(i)) has been skipped, as this is the node in which the slowscan starts off. This length can be described as

\[ \|\text{head}(i + 1)\| - \|s(\text{head}(i))\| = \|\text{head}(i + 1)\| - \|\text{head}(i)\| + 1 \]

Thus, the sum over all these lengths \( \|\text{head}(i + 1)\| - \|\text{head}(i)\| + 1 \) for \( i = 1..n \) must equal the number of character-wise comparisons performed by slowscan during all \( n \) iterations of the algorithm. Calculating this sum gives

\[
\begin{align*}
\|\text{head}(2)\| - \|\text{head}(1)\| + 1 \\
+ \|\text{head}(3)\| - \|\text{head}(2)\| + 1 \\
+ \cdots \\
+ \|\text{head}(n + 1)\| - \|\text{head}(n)\| + 1 = \\
\|\text{head}(n + 1)\| - \|\text{head}(1)\| + n = n
\end{align*}
\]

as \( \|\text{head}(n + 1)\| \) and \( \|\text{head}(1)\| \) both are empty. As suffix \( n + 1 \) is the empty string \( \lambda \), then so is head(n + 1), and there is no suffix prior to the 1st suffix with which it can have an LCP of any positive length, so head(1) must too be empty by the definition of head. This concludes, that the total number of comparisons performed by slowscan during all \( n \) iterations of the algorithm is \( n \) and thus within \( O(n) \).

Time complexity of Fastscan

Fastscan is characterized by scanning down the tree much faster than slowscan, specifically in time \( O(1) \) per downward edge walked, rather than time proportional to the combined length of the edges scanned during slowscan.

As fastscan will only ever walk downwards in a tree, it is bounded by the maximum depth of the tree. However, whenever fastscan is used, the depth is first decreased by \( 1 \) when consulting the parent of the node, then potentially another depth decrease of \( 1 \) by following the suffix link from the parent node, as argued in Section 2.8. This sums to \( 2n \) potential depth decreases over all \( n \) iterations of the algorithm.

After each depth decrease, fastscan proceeds to walk down the tree to cover what is currently being scanned after. It may, alongside slowscanning, scan all the way towards the maximum depth of the tree,
which is potentially $n$ for a suffix tree of a string of length $n$, as stated in Section 2.6. Although the worst-case per iteration of fastscan may look as though fastscan could potentially be an $O(n^3)$ procedure all in all, as it could potentially scan down the whole depth of the tree in a single iteration, an amortized analysis of the procedure concludes otherwise. As both fastscan and slowscan will always be followed by an addition of a node to the tree and thereby potentially increase its depth to no more than a maximum total depth of $n$ of the tree, and as fastscan will decrease the depth by no more than 2 levels before increasing it again by walking down, the fastscan procedure will be bounded by $3n$ edges walked over the course of all iterations of the algorithm. This makes fastscan perform within time $O(n)$.

**Alphabet size reliance**

As the algorithm relies on determining the edge to pass down along during both scanning strategies, it must compare the initial characters on all downward edges of a node of which there can be up to $|\Sigma|$ of, where $\Sigma$ is the ordered alphabet from which $S$ was drawn. Consider the following implementation: At each node in the tree, maintain a list of length $|\Sigma|$. For each entry in the list, let it correspond to a symbol $a$ in $|\Sigma|$, and let the entry point towards a child, if any, having initial character $a$ on the edge going down to this child in the tree. Now, determining which child, if any, has a downward edge leaving an internal node with a label starting with character $c \in |\Sigma|$, a binary search in this list will find the entry corresponding to $c$ in time $O(\log |\Sigma|)$. As this potentially happens in each internal node, of which there can be $O(n)$, the algorithm has a total running time of $\Theta(n \log |\Sigma|)$. Thus, an alphabet of linear size would result in a running time of $\Theta(n \log n)$, which is why the linear running time performance only holds under the assumption that the alphabet is of constant size, which is a typically made assumption.

Farach’s algorithm, which we will see in Section 6.1, does not rely on determining which edge to pass down along when looking for the location of some suffix. Instead, it utilizes radix sort or similar linear time, stable sorting algorithms, and build the tree off of those. This saves the $\log n$ factor in the running time complexity on alphabets of linear size, such as integer alphabets. The result is a linear time algorithm for alphabets of linear size.

Before delving into that algorithm, however, another algorithm for quick retrieval of the LCA of two nodes, on which the algorithm by Farach relies, will be explained in the following section.
EFFICIENT LOWEST COMMON ANCESTOR RETRIEVAL

In the construction of the suffix tree, we need to find lowest common ancestor in constant time. The article used to implement Farach’s short mention that it is possible, and references article [13]. The details wasn’t elaborated further, which we will now do.

We will now explain LCA retrieval as described in [11]. Whenever the details of the article was unclear to us, the explanation from [23] was used as supplement.

5.1 PREPROCESSING

In the construction of the suffix tree, we need to find lowest common ancestor in constant time. We can do this with a simple trade off, we need to preprocess the tree. We will now show how to preprocess the tree in linear time, then we will show how to find the lowest common ancestor in constant time.

We will start by explaining the theory, and then show how it works in practice. The theory builds around constant retrieval in a binary tree, which can be expanded to constant retrieval in trees in general. We will explain these steps fully, even though it is not used in practice. We choose to explain both approaches, as the steps needed in practice is not intuitive without further explanation. When describing the algorithm we will call the original tree T and the binary tree B.

The binary tree

We will start by looking at a rooted complete binary tree, B. B has l leaves, and a total height of \( \log_2(l) \) which we will refer to as \( h \). Each node in B is assigned a bit number consisting of exactly \( h + 1 \) bits, which we will refer to as the path number. The path number of a node \( v \) is describing the path from the root node to \( v \). Each bit \( i \) in the path number describes whether the path to \( v \) went left or right in a node at height \( i \) in B, with \( i = 0 \) corresponding to a path going left at height \( i \), and \( i = 1 \) corresponding to a path going right at height \( i \). Example in Figure 11 the path to node 5, is 101. Bit 3 (from the right), is one, indicating a right turn which is evident from the edge it went down from root node. In the next node, with height 2, it took a left edge which is indicated with 0.
Constant LCA retrieval in Binary Tree

For two nodes $x$ and $y$, we will denote the lowest common ancestor $\text{lca}(x, y)$.

When finding $\text{lca}(x, y)$, we have 2 cases.

The first case

In the first case either $x$ or $y$ is an ancestor of the other, $x$ or $y$ must then be the $\text{lca}(x, y)$. We can easily check if either $x$ or $y$ is an ancestor to the other. We first start by post-order traversing the tree, denote the order in which the nodes are visited. This results in a tree like in Figure 12. After traversing the tree we can check if a node $x$ is an ancestor of a node $y$. For this to be true, two things need to be true. First the post-order numbering of $x$ should be lower than that of $y$. Second the post-order numbering of $y$ should be lower than the numbering of $x$ + descendants($y$), where descendants($y$) describes the number of child’s below $y$ including the node itself. If both conditions hold, $x$ must be $\text{lca}(x, y)$. The same should be checked for $y$. Example, in Figure 12 the node 2 is an ancestor of node 3. First we compare the two nodes post-order numbering $2 < 3$, then we test the second condition, we can see on Figure 12 that 3 has 3 descendants (including itself), so we check $3 < 2 + 3$. Both conditions are satisfied and we conclude that 3 is indeed an ancestor of 2 and must be their lowest common ancestor.

The second case

In the second case, neither $x$ or $y$ is the lowest common ancestor, which we will assume from now on.

By our definition of the path number, we know that two nodes $x$ and $y$ that are descendants of a node $z$ with height $h(z)$, must share the same path number on the left-most first $h(z)$ bits as they share the path to $z$. 
To find \(x\) and \(y\)'s shared left-most bits, we first \(\text{xor}\) the two, the first left-most one bit is the height from where the two nodes no longer share path. We call this position \(k\).

Example from Figure 11, \(x = 101\) and \(y = 111\). \(x \text{xor} y = 010\), the \(k\) left-most bit is found on position 2, telling us that the two nodes must share path until they reach a node with height 2 which, in this case, they do, as they both descent directly from node 6 with height 2.

We can now find their lowest common ancestor by filling the bits left of position \(k\), with either the \(k-1\) left most bits from \(y\) or \(x\), which we already found was shared. Position \(k\) is set to 1, and the right side bits should all be set to 0. The resulting bit number is the result of the two nodes shared path in the binary tree. This number is exactly the same as their lowest common ancestor. Continuing the same example as before, we create a new bit number \(b = 000\). We now set the \(k\)'th bit in \(b\) to 1, \(b = 010\). We now have to fill the bits left of \(k\), with the corresponding bits of either \(x\) or \(y\), this results in the path number for the \(\text{lca}(x, y)\), \(b = 110\). \(\text{lca}(5, 7) = 6\).

**Mapping to binary tree**

We now know that we can find \(\text{lca}(x, y)\) in the binary tree in constant time, the original tree is not necessarily a binary tree. We will now look at a mapping from our original tree, \(T\), to a binary tree, \(B\).

We start by denoting every node with a depth-first search number, we will call this number \(\text{PREORDER}\). \(\text{PREORDER}\)'s for our tree can be seen in Figure 13.

We will refer to a node in \(T\) by its \(\text{PREORDER}\) number. The index of the least significant bit of the \(\text{PREORDER}\) in node \(v\) will be referred to as \(\text{lsb}(v)\). Example for 6 with binary representation 110 is \(\text{lsb}(6) = 2\).

After denoting all the nodes with its corresponding \(\text{PREORDER}\), we need to traverse the tree starting at the leaf nodes and moving up.
Figure 13: Example tree $T$, with traversal for denoting the PREORDER onto the nodes. The structure of this tree will be used as an example throughout the rest of this chapter.

Figure 14: The tree $T$ with the binary representation of their PREORDER. Each node, $v$, has a pointer to $I(v)$. Each run is marked with a box around the nodes contained in a run.

along parent edges. When traversing we will, for each node $v$, keep track of the $\text{lsb}(v)$, if $\text{lsb}(v) \geq \text{lsb}(w)$, where $w$ is all the preceding nodes, we will denote this $\text{lsb}(v)$ to be the currently largest and set $v$'s INLABEL to $v$ we will refer to the INLABEL of a node as $I(v)$. If $\text{lsb}(v)$ is smaller then a preceding $\text{lsb}(w)$, then we set $v$’s INLABEL to the previous node’s INLABEL. After the tree is traversed we will end with a tree, where each node has a reference to the node in its sub-tree with the largest $\text{lsb}(v)$. A collection of nodes pointing to the same INLABEL, we call a run, and can be seen in Figure 14.

We will now define a head of each run, to be the node in a run with the lowest depth. We will make a mapping from all the INLABELs to the head of each run, we call this mapping $L$. We can now find the head of the run containing node $v$, in constant time, by looking up $L(I(v))$. 
Figure 15: The tree T denoted it ancestor list, Av.

Figure 16: Binary tree for our tree in Figure 14.
Note that if a node \( w \) is an ancestor to a node \( v \), then \( L(I(w)) \) must either be an ancestor of \( (I(v)) \) or be \((I(v))\).

After we have created the runs, we can make the mapping to a binary tree initialized as described earlier with a height of \( \log_2(n) \), as seen in Figure 16, where \( n \) is the number of nodes in \( T \). We map all the unique INLABEL’s to a node with the same path number.

While mapping to the binary tree, we attach a bit number to each node, describing a node’s ancestor. We will call this \( A_v \) for a node \( v \) and should have the same number of bits as the total height of the binary tree, \( O(\log n) \). Each bit in \( A_v \) denotes whether the node has an ancestor, if the \( i \)’th index is 1, then node \( v \) has an ancestor in height \( i \), \( A_v \) for the example can be seen in Figure 15. Example if we look at Figure 14, and iterate throw node 0110’s ancestors we find that there are 3 different runs, 1000, 0100 and itself 0110. 1000 is at height 3 in the binary tree, 0100 is at height 2 and 0110 is at height 2. The ancestor bits of node 0110 must then be 1110

**Implementation**

In the actual implementation, we don’t need the binary tree, when retrieving lowest common ancestor in constant time, we only need to know they height of \( I(z) \), where \( z \) is the lowest common ancestor. In practice we only need to do the following steps:

1. Make a depth first search and give all nodes their PREORDER number. While doing this, create a look-up table to find every node given their PREORDER.

2. Traverse through every node \( v \) in the tree, setting \( I(v) = v \) and checking for each \( v \) whether \( \text{lsb}(I(w)) \geq \text{lsb}(I(v)) \) holds for any children \( w \) of \( v \), if true set \( I(v) \) to \( I(w) \) for the child with the greatest \( \text{lsb}(I(w)) \).

3. Traverse down through the tree, keeping track of a \( A_v \). For each node \( v \), first set \( A_v \) to \( A_w \) where \( w \) is the parent of \( v \). Second update \( A_v \) and set the \( i \)’th bit to 1 where \( i = h(I(v)) \).

When looking at the three steps necessary, it is quite clear that the preprocessing is in \( O(n) \) time, as we run through the tree exactly 3 times.

As showed, after preprocessing we can find the height of two nodes INLABEL, this can now be used to find the lowest common ancestor in a tree, which we will show now.
5.2 LCA QUERIES IN CONSTANT TIME

Given two nodes $x$ and $y$, from a tree which was preprocessed according to section 5.1, we want to find their lowest common ancestor $z$ in constant time.

We start by looking up $I(x)$ and $I(y)$. By the constant-time lookup described in section 5.1, we find the lowest common ancestor of $I(x)$ and $I(y)$ in the binary tree. We call this node $b$.

Based on the height of $b$ we can deduce the height of $I(z)$. We do this by first finding the least significant bit, greater than the height of $b$, that both $A_x$ and $A_y$ shares. We call this $j$, and is the height where we can find $I(z)$.

Now we know the height of $I(z)$, we want to find the node closest to $x$ and on the same run as $z$. We do this by first finding the right-most 1-bit in $x$’s ancestor list. If this equals the $j$ we found earlier, $x$ must already be in the same run as $z$.

If not, we can now create a new bit number consisting of 0’s to the right of $j$, and consists of the same bits as $I(x)$ to the left of $j$ and set the $j$’th bit to 1. This will result in a number matching a node, which we will call $w$. When taking the head of the run $I(w)$, we have a node directly under the run $I(z)$. We now find the parent of $w$. This node must be in the same run as $z$, we call this node $x'$.

We can now do the same for $y$, and we have found the closest node $y'$ from $y$ that is in the same run as $z$.

Now we have found both $y'$ and $x'$, the one with the smallest PREORDER must be the lowest common ancestor of $x$ and $y$.

**Implementation**

1. Find the left-most $i$’th bit where $x$ and $y$ differ. Set a variable $h$, indicating the height of $I(z)$, to be the maximum of either $i$, the least significant bit of $x$ or the least significant bit of $y$.

2. Find the least significant bit that $A_x$ and $A_y$ share, where the bit should be greater or equal to $h$ just found. Call it $j$.

3. Find the right most bit $i$ in $x$’s ancestor bit-list. If $i$ is equal to $j$, then skip step 4. Repeat for $y$.

4. Find the left-most 1-bit in $A_x$, to the right of $j$. Create a new bit number, fill the bits left of $j$ with the bits corresponding in $I(x)$. Set the $j$’th bit to 1, and fill the right of $j$ with zeros. The parent to this bit must be a node in the run $I(z)$. We set $x'$ to be this node.

5. Set $z$ to be the node of $x'$ and $y'$ with the lowest PREORDER.
We have now shown, that with only preprocessing the tree with PREORDER, INLABEL and runs, we can find LCA of two nodes in constant time.

Figure 17: Example of the explained steps, finding LCA(3, 6).
In the following, a description of the algorithm by Farach [6], will be given which, given an input string over an integer alphabet, will construct a suffix tree in linear time.

The algorithm was later presented in [25, Ch. 5.2.4] as well as [26, p.65-72] in a more pedagogical manner, focusing on implementation specific details. Our explanation will follow the course of the presentation given in [25], but will be supported by a functioning implementation.

In the process of implementing the algorithm, uncertainty as to what was meant in this presentation arose at times. As a result, our explanation will focus on these aspects of the algorithm, elaborating how we chose to handle them through detailed figures and pseudocode, and as such further aid the process of implementing this algorithm for anyone who would wish to do so themselves.

6.1 Algorithm Overview

Initially, the input string $S$ is split into two smaller inputs. This division, as part of the divide and conquer strategy utilized in the algorithm, will yield two suffix trees from a recursive call, namely $T_{\text{odd}}$ and $T_{\text{even}}$, of all suffixes found at odd and even positions, respectively. This is explained in Chapters 6.2 and 6.3.

These two trees are then overmerged in a way that ensures that a merging of subtrees of $T_{\text{odd}}$ and $T_{\text{even}}$ does in fact happen in all places needed, but the procedure may merge more than what is correct, giving us an overmerged tree $M_S$ over $S$. The process of merging is explained in Chapter 6.4.

As a consequence of the overmerging, an adjustment procedure is necessary to correct all places that were erroneously merged. The necessary preprocessing of $M_S$ is explained in 6.5, and the actual adjustment procedure is explained in 6.6. The result of this sequence of procedures is a complete and correct suffix tree $T_S$ of $S$.

The explanations of the above subprocedures will each be followed by their time complexity. Chapter 6.8 will then lastly convince that the algorithm as a whole is within the alleged time complexity of performance of the algorithm, totaling a suffix tree construction algorithm running in linear time.

For an implementation of the algorithm, to aid in the understanding hereof, we will refer to our own, which can be located as described in Section 7.2. During the development of said implementa-
tion, we made choices different from those made in the presentation of the algorithm as given in \cite[Ch.5.2.4]{25}. These differences are outlined throughout our presentation of the theory, and are further discussed in Section 6.7.

### 6.2 Constructing the Odd Suffix Tree $T_{\text{odd}}$

For the construction of the odd suffix tree $T_{\text{odd}}$, as introduced in Section 6.1, recall the assumption that $S$, the input string on which the construction of $T_{\text{odd}}$ will be based, is a string over an integer alphabet. This is not an unreasonable assumption, as any string over any alphabet can be converted to an equivalent string over an integer alphabet in a single iteration. An example of such an integer string is

$$S = 12323232\$$$

Note that the $\$ symbol in practice is an integer too, higher than any other integer in the string, but is preserved as $\$ throughout the explanation of the algorithm to emphasize its particular role as the last character.

Initially, a suffix tree of all the suffixes found at odd positions in $S$, from here onwards referenced as odd suffixes, is to be constructed. Such a suffix tree will be denoted $T_{S_{\text{odd}}}$.

Initially, the string $S$ is split up into pairs of characters such that the odd indexed characters in the string are paired with the following even indexed character, creating a list of character tuples

$$(S[1..2], S[3..4], S[5..6], \ldots, S[n-1..n])$$

For example, when applied to $S$, the following pairs are found

$$[(1, 2), (3, 2), (3, 3), (2, \$)]$$

The resulting list contains tuples of characters on the form $(x, y)$. This list is then sorted using radix sort, first sorting on $y$ and then $x$. This sorting takes $O(n)$ using radix- or bucket sort, as the id’s are all in range $1..n$. After sorting, duplicates are removed, resulting in the following list

$$[(1, 2), (2, \$), (3, 2), (3, 3)]$$

This list is now lexicographically sorted and contains unique pairs of characters. Based on the sorting, every pair is now assigned a rank corresponding to their occurrence in the sorted list, such that first pair in the list gets rank 1, second pair rank 2 etc. Whether considering
6.2 Constructing the Odd Suffix Tree

Figure 18: Suffix tree, $T_{S'}$, of the string $S'$. Figure 19: $T_{S'}$ extended, the nodes now match the original string, $T_{S'}$ is no longer a correct compacted trie, as multiple edges share first character.

ranks or the lexicographic relation between characters is now a matter of preference as the ranks express just this, lexicographic relations.

\[
\text{rank} = \begin{cases} 
(1, 2) \rightarrow 1 \\
(2, $) \rightarrow 2 \\
(3, 2) \rightarrow 3 \\
(3, 3) \rightarrow 4 
\end{cases}
\] (3)

By replacing each character tuple in (2) with its corresponding rank, as determined by (3), a new string $S'$ of ranks is formed

\[S' = 1342\]

Calling the full suffix tree construction procedure recursively upon $S'$ yields a suffix tree $T_{S'}$, as illustrated in Figure 18.

This tree is obviously not the suffix tree of the odd suffixes that is intended, rather this is a tree of a related string, $S'$. To derive the odd suffix tree for the original string $S$, $T_{S'}$ must be adjusted. Recall that each character in $S'$ represents a character pair in $S$, so each string on an edge in $T_{S'}$ must be twice as long in $T_S$. Furthermore, the suffix id's must be altered in a similar way, as $S$ is twice the length of $S'$, thus the $i$'th suffix in $S'$ must correspond to suffix $2i - 1$ in $S$. Applying these changes, however, is not sufficient, as the resulting tree is not necessarily a compacted trie any longer, as is evident from Figure 19.

When modifying the string $S$ to the string $S'$, character pairs were replaced with their corresponding rank. Because two character pairs can share the first character, but never both (e.g. in the pairs $(3, 2)$...
and (3, 3)), the tree $T_{S'}$ risks being an incorrect compacted trie with multiple outgoing edges from a single node sharing the same initial character.

**Time complexity of constructing $T_{odd}$**

This is handled by going through every node in the tree, examining whether any of its outgoing edges for some number of siblings starts with the same character. As $T_{S'}$ is assumed to be sorted, siblings sharing initial characters will all be neighbours. If identical initial characters is the case, a new internal node must be added, and all the siblings whose edges started with the same character is then added as children to this new internal node. When inserting the siblings, the lexicographic ordering of the tree must be preserved, so the second character, on which they all are guaranteed to differ for different ranks, determines the order of insertion here.

The ranking of the character tuples and the constant time modifications performed in potentially each node and edge in $T_{S'}$ are each asymptotically bound by $\Theta(n)$. The time complexity of the recursive call will be discussed later. After these adjustments the resulting tree will be $T_{S_{odd}}$, the suffix tree of all odd suffixes. One such is illustrated in Figure 20.

### 6.3 Constructing the Even Suffix Tree $T_{even}$

The suffix tree $T_{even}$, as introduced in Section 6.1, is a suffix tree for suffixes found at even positions in the input string, from here onwards referenced as even suffixes. Having constructed $T_{odd}$ in the previous step, $T_{even}$ can be constructed on the basis of $T_{odd}$ as the following explanation will show.

**Even suffix ordering**

The lexicographic ordering of odd suffixes is evident from their occurrences in a depth-first traversal of $T_{odd}$, as $T_{odd}$ is sorted by construction. As an even suffix is simply a single character followed by an odd suffix, the lexicographic ordering of the even suffixes can be deducted from this traversal of $T_{odd}$ as follows: Pairing all the sorted
6.3 constructing the even suffix tree $T_{even}$

Odd suffix id’s from the traversal with the character immediately in front of them will yield tuples on the form

$$(S[2i], 2i + 1)$$

If this ordered list of tuples is then sorted using radix sort with respect to this newly prefixed character $S[2i]$, a lexicographically sorted list of the even suffixes will result when the odd id’s $2i + 1$ are subtracted by 1.

An important attribute of radix sort is its stability. It will maintain the initial order of the tuples, and only rearrange according to the even character, leaving identical even characters to be sorted by the known order of the following odd suffixes.

Note that if the last character, which is the unique symbol, is found at an even position in $S$, the last suffix will be an even one, and it will only consist of the unique character which, by definition, is the last symbol in the lexicographic ordering of the alphabet, i.e. the highest occurring integer when translated. Therefore, it will be correct to just append it to the end of the sorted list of even suffixes afterwards.

**Determining the LCP of even suffixes**

By preprocessing $T_{odd}$ as described in 5, the LCP for all odd suffix pairs can now be determined, as it will correspond to the LCA in suffix trees per Section 2.7. As stated above, this also gives the LCP for even suffixes $2i$ and $2j$ where $i, j \in 1..\lceil n/2 \rceil - 1, i \neq j$ by

$$LCP(2i, 2j) = \begin{cases} LCP(2i + 1, 2j + 1) + 1 & \text{if } S[2i] = S[2j] \\ 0 & \text{otherwise} \end{cases}$$

We now have an ordered list of the even suffixes, and the length of the LCP of neighbouring suffixes in this ordered list can be determined as well. The even suffix tree $T_{even}$ must now be constructed in a way such that a depth-first traversal through the tree will visit leaf nodes representing the suffixes in an order matching this sorted list.

**Constructing $T_{even}$**

Knowing this order in which leaf nodes are visited along with the length of their pairwise shared paths from the root node, i.e. their LCP, $T_{even}$ can be constructed. This is done from left to right, one node at a time. Each node $v$ is inserted on the path from the root node to the most recently inserted node $v'$ at a position matching the $LCP(v, v')$. If this position is in between two nodes $u$ and its parent $p(u)$, i.e. on the edge between the two, a new node $u'$ is created and inserted at this position below $p(u)$ whereafter $u$ and $v'$ will both be attached as children to this new internal node, $u'$. An illustration of this procedure is shown in Figures 21 and 22.
Figure 21: The path from the root node to the previously inserted node $v'$ onto which each new node $v$ should be inserted.

Figure 22: In (1), the position of $v$ was determined to be partway down the edge between two nodes, $u$ and $p(u)$. A new internal node $u'$ is placed here, as shown in (2), of which $v$ can then branch off, and the algorithm proceeds to consider a new node in (3) with $v$ now being the previously inserted node.
Time complexity of constructing $T_{even}$

The depth-first traversal of $T_{odd}$ takes $\Theta(n)$, and the radix sort on the characters are assumed to be bound by $O(n)$, too. The construction of $T_{even}$ itself, by insertion of each of the leaf nodes $v$ on the path from root to the most recently inserted leaf node $v'$, is bound by $\Theta(n)$ as well. This is so, as a decision to either insert a new leaf node or skip an entire subtree is made in potentially each internal node. Thus, every internal node may be considered multiple times, but each such consideration will lead to either skipping the internal node and not revisiting anymore, or inserting one of the $n/2$ leaf nodes below it. The result is a correctly constructed and lexicographically sorted suffix tree $T_{even}$ of all even suffixes in $\Theta(n)$ time.

6.4 OVERMERGING OF $T_{odd}$ AND $T_{even}$

Now that $T_{odd}$ and $T_{even}$ have been correctly constructed, it is time to merge them into a single tree $M_S$, as introduced in Section 6.1.

As the trees are sorted by construction, pairs of children of the root node in both trees will be considered from left to right while concurrently constructing $M_S$ from left to right also. When comparing two
Figure 25: Case 1, node 7 is inserted the first character differs, and can be inserted lexicographically in the current node.

children from $T_{odd}$ and $T_{even}$, respectively, for insertion in $M_S$, the strategy is to consequently overestimate commonalities between the two, and then later, when the structure is in place, adjust down these overestimations to encapture actual commonalities. The length of and the first character in the represented string of each of the two nodes are available for comparison, thus one of three cases will apply in any such comparison of these nodes:

1. The first character in the string differs
2. The strings have matching first character, but are of different lengths
3. The strings have matching first character and lengths

Case 1

The first case is the simplest of the three. As nothing is common between the two strings, nothing needs to be merged, and all that will matter in this case is making sure $M_S$ is constructed in way that leaves it sorted. It can be inferred from the first character alone which of the nodes should be inserted into $M_S$ at the current position, as the alphabet consists of integers whose natural order is known.
6.4 overmerging of $T^{odd}$ and $T^{even}$

Case 2

The second and third cases are a bit more cumbersome, as a proper string of unknown length is shared which will require some type of merging on top of ensuring a sorted outcome. Both of these cases will introduce new internal nodes which will be referred to as overmerged nodes. The two nodes being merged will be referred to as $m^{odd}$ and $m^{even}$.

In the second case, the nodes share said string of some positive length, and obviously no longer than the shorter of the two strings. It is now assumed that the shorter node is, in its entirety, a proper prefix of the longer node. A likely to be wrong claim, but corrections are postponed until as well as explained in Section 6.6. Under this assumption, the case is handled by creating an internal node, the overmerged node, in $M_S$ representing the shorter of the two suffixes of the input string, even though these are assumed to be all leaf nodes. This internal node representing a suffix will also be handled later in the adjustment of the overmerge. As we might be overmerging two subtrees containing more than just $m^{odd}$ and $m^{even}$ in this second case, we need to recursively merge these, as we have just added another node to $M_S$ whose subtree consists of nodes from both trees $T^{odd}$ and $T^{even}$.

Case 3

As for the third case, the two nodes $m^{odd}$ and $m^{even}$ share at least one character, but might be entirely identical given that they are of equal length, and the fact that we look no further than the first character. The assumption once again is identicalness among the two strings, thus merging the two nodes into one in $M_S$ is a reasonable action to take under this assumption. This is done by creating a new node $m'$ to represent both $m^{odd}$ and $m^{even}$.

Figure 26: Case 3, the strings have matching first character, but are different lengths
Figure 27: Case 2, the two strings match on first character and have same lengths. This will result in a recursive merge of the two sub trees.

As with the previous case, \( m' \) may contain nodes from both \( T^{\text{odd}} \) and \( T^{\text{even}} \) in its subtree besides \( m^{\text{odd}} \) and \( m^{\text{even}} \), and as such needs to be overmerged recursively as well to ensure proper merging and lexicographic sorting of the subtree of \( m' \). Note as well, that a similar structure to the previous case is possible, where a leaf node in one tree shares length and initial character with a subtree in the other tree. In this case, the node that the two subtrees are merged into must be annotated with the id corresponding to the leaf node in order to preserve information of this particular suffix in the overmerged tree.

As to prepare for later steps in the algorithm, it is practical to note that whenever nodes are being overmerged, it is done to represent some string shared by at least two nodes, one from each tree \( T^{\text{odd}} \) and \( T^{\text{even}} \). This results in the overmerged node being the LCA for some odd/even leaf node pair, and such a pair is needed in every overmerged node later, which is done most easily during the actual overmerge. In the explanation given in [25], these node pairs are identified based on the order in which leaf nodes are visited in a depth-first search through \( M_S \), but we were not able to reconstruct this. Instead, through proper bookkeeping, one can ensure having taken note of one odd and one even leaf node in the subtrees of two distinct children of each overmerged node during the construction of \( M_S \). These odd and even leaf nodes will have said overmerged node as their LCA. More on the explanation in the book and our alternative implementation is found in Section 6.7. Also, a reference in each overmerged node to the nodes from \( T^{\text{odd}} \) and \( T^{\text{even}} \) that was merged
Figure 28: $M_S$ is the result of overmerging $T_{\text{even}}$ and $T_{\text{odd}}$. Due to the assumptions in cases 2 and 3 (here, case 3 handled the overmerge), $M_S$ has an edge of length 2 and a subtree that is the result of recursively merging $T_{\text{even}}$ and $T_{\text{odd}}$ below this edge. Here, the correct merge is an edge with only length 1, and the remaining differing character, here “a” for $T_{\text{odd}}$ and “b” for $T_{\text{even}}$, being appended to the subtrees of each. This shows, that $M_S$ may need adjustments before being correct.

in that particular overmerge will be practical later, so annotation of these during cases 2 and 3 is advised as well.

**Time complexity of constructing $M_S$**

Visiting potentially every node in both $T_{\text{odd}}$ and $T_{\text{even}}$ totals a number of nodes within $\Theta(n)$. Each of these nodes will be subject to comparisons of a single character and the length of the string represented by that node. A such comparison may result in the creation of a new internal node. These are all $O(1)$ time operations, and so is the bookkeeping advised for later steps, making the overmerging procedure be within $\Theta(n)$.

6.5 **LCP tree construction**

As a consequence of the assumptions in cases 2 and 3 in the construction of the overmerged tree $M_S$, the nodes in the tree might not have been merged properly and thus needs adjustments as is evident from Figure 28.

**Identifying faulty overmerges**

Let $2i$ be a leaf node from $T_{\text{even}}$ and let $2j - 1$ be a leaf node from $T_{\text{odd}}$. Then the overmerged node $v = \text{LCA}(2i, 2j - 1)$ is a node in which a
potentially faulty overmerge has happened. A correctly performed merge is characterized by the relation in Equation (4), summarized from its first occurrence in Section 2.7.

\[
\text{LCA}(2i, 2j - 1) = \text{LCP}(2i, 2j - 1)
\]  

(4)

The left side of this equation corresponds to the current length of the strings of overmerged nodes, and the right side is the desired lengths of said nodes’ strings. An incorrect merge will have \(\text{LCA}(2i, 2j - 1) > \text{LCP}(2i, 2j - 1)\). When the relation expressed in Equation (4) holds for all internal nodes \(v\) in \(M_S\), it is a correct suffix tree, thus some means of checking this relation in an overmerged node is necessary.

**Testing Equation 4 in nodes**

Each overmerged node \(v\) is easily identifiable, as they were all annotated with a descending odd and even leaf node during the overmerge in our implementation. To determine \(\text{LCP}(2i, 2j - 1)\), recall from 6.5 that the length of the LCP of two nodes is equivalent to the depth of their LCA in the LCP tree, so the construction hereof will allow for looking up \(\text{LCP}(2i, 2j - 1)\) to verify Equation (4).

**Creating suffix links**

Let \(s_i = S[i..n]\) and \(s_j = S[j..n]\) be two suffixes such that \(\text{LCP}(s_i, s_j)\) has length \(m\), i.e. \(\text{LCP}(s_i, s_j) = S[i..i+m] = S[j..j+m]\). The two suffixes following \(s_i\) and \(s_j\), namely \(s_{i+1} = S[i+1..n]\) and \(s_{j+1} = S[j+1..n]\) must then have an LCP \(\text{LCP}(s_{i+1}, s_{j+1})\) of length \(m - 1\), specifically \(\text{LCP}(s_{i+1}, s_{j+1}) = S[i+1..i+m] = S[j+1..j+m]\). This keeps going until nothing is shared any longer, i.e. until the length \(m\) reaches 0, as illustrated in Figure 29. Recall the definition of suffix links. The node representing \(\text{LCP}(s_i, s_j) = S[i..i+m]\) must have a suffix link pointing to the node representing \(\text{LCP}(s_{i+1}, s_{j+1}) = S[i+1..i+m]\). These nodes are \(\text{LCA}(i, j)\) and \(\text{LCA}(i+1, j+1)\) respectively. This means that suffix links should be created such that the link from \(\text{LCA}(i, j)\) points towards \(\text{LCA}(i+1, j+1)\).

As Equation (4) must be tested in all overmerged nodes, these should all have a suffix link. Determining where to point it to is merely a matter of looking up \(\text{LCA}(2i+1, 2j)\), i.e. the suffixes following the two for which \(v\), the overmerged node, was the LCA. The two nodes for which \(v\) is the LCA was annotated on \(v\) during the overmerge in our implementation, yet in the presentation of Farach’s algorithm given in [25], they accomplish this through a nifty analysis of the occurrences of leaf nodes in a depth-first traversal of \(M_S\). We were not able to reproduce this identification of nodes, and resided to
identify them during the overmerging. This analysis, our issues and our solution will be discussed in Section 6.7.

The assumption that nodes can be accessed in constant time given their id is required here to retrieve the leaf nodes \( i+1 \) and \( j+1 \). Further, the tree must be preprocessed according to Section 5 as to allow for constant time retrieval of the node \( \text{LCA}(2i+1, 2j) \) given nodes with id’s \( 2i+1 \) and \( 2j \).

Once the overmerged nodes and their corresponding leaf node pair for which they are the LCA has been identified, and \( M_S \) has been preprocessed to allow for constant time retrieval of the LCA of the following pair of suffixes, the construction of the LCP tree is a straightforward process, possible in a single traversal of \( M_S \).

**Practical note**

In practice, however, only the minimum size subtree of the full LCP tree that contains all the overmerged nodes are relevant. These are the only nodes in which the depth is used, as they are the only nodes susceptible to errors from the overmerging of \( T^{\text{odd}} \) and \( T^{\text{even}} \), as exemplified in Figure 30. Luckily, the suffix link of all overmerged nodes will point to another overmerged node. This is the case as all overmerged nodes \( v \) are the LCA of an even and an odd leaf node, with id’s \( 2i \) and \( 2j-1 \) respectively. The suffix link must then, as established earlier, point to the LCA of \( 2i+1 \) and \( 2j \), which must also be odd and even respectively, making their LCA a node for which the subtree contains both odd and even nodes, i.e. an overmerged node. This boils down the construction of the LCP tree to only building suf-
Figure 30: The figure shows two suffix links built atop the overmerged tree $M_S$ of $S = 12313\$$. Constituting the full LCP tree of $M_S$, however, only the suffix link marked in bold from node 4 is relevant as node 4 is the only overmerged node, thus the only candidate for adjustments.

Figure 31: The node visited initially by the breadth-first traversal has a depth in the LCP tree one larger than the depth of the node visited second. As the second node has not yet been visited, its depth is unknown, and thus the depth of the first node cannot be determined at this point. A breadth-first traversal alone is not sufficient to annotate all nodes with their depths. We solved this using recursion during the traversal, as sketched in Figure 32.

Fix links on the subset of the internal nodes which is the set of all overmerged nodes.

**LCP tree depth annotation**

When the LCP tree has been constructed, annotating all its nodes with their depths in this tree can be done in a breadth-first search of $M_S$, starting with the root node which has depth 0 and then letting the depth grow downwards, such that each node $v$ have a depth which is one larger than the depth of the target of its suffix link. Note, however, that suffix links may point ahead of the breadth-first search, so a depth might not be available at the end of the suffix link yet, on which the depth of the current node should be based upon. This can be handled by a recursive procedure such as the one sketched in Figure 32. In Figure 29, the resulting LCP depth annotations are shown on an example tree.
function calc_lcp_depth(node):
    if hasattr(node, 'lcp_depth'):
        # This node was previously visited, return its depth
        return node.lcp_depth

    lcp_tree_parent = node.suffix_link
    if not hasattr(lcp_tree_parent, 'lcp_depth'):
        # suffix link points ahead of bfs; call recursively
        node.lcp_depth = calc_lcp_depth(lcp_tree_parent) + 1

    node.lcp_depth = lcp_tree_parent.lcp_depth + 1
    return node.lcp_depth

root.lcp_depth = 0
bfs(overmerged_nodes, calc_lcp_depth)

Figure 32: Pseudocode for annotating overmerged nodes with their corresponding depths in the LCP tree using a breadth-first search and recursion

This leaves $M_S$ with all overmerged nodes annotated with their depths in the LCP tree, which equals the desired length of the string represented. All that is left is to adjust $M_S$ around nodes in which the relation in Equation (4) does not hold.

Time complexity of the LCP tree construction

Producing the LCP tree itself was essentially a matter of constructing suffix links on overmerged nodes. Deciding where to point these was done in $O(1)$ due to the $O(n)$ preprocessing performed on $M_S$. The annotation of depth was essentially a breadth-first search through $M_S$. As no more than $\Theta(n)$ overmerged nodes will exist, the entire construction is therefore bound by $\Theta(n)$.

6.6 Final Adjustments

The creation of the LCP tree allows for identification of faulty overmerges by comparing represented string length with the LCP depth in each overmerged node. Not only that, it also states exactly to which extent the overmerging was wrong if so. What is needed is essentially a partial unmerging of the overmerge.

The process of unmerging

Let $v$ be a such overmerged node in which an error has been identified. This node is the result of overmerging nodes $m_{odd}$ and $m_{even}$ from $T_{odd}$ and $T_{even}$, respectively, to which references were created in $v$ during the overmerging of said nodes.
Figure 33: An example of a faulty overmerge on string $S = \text{"ababcacac$"}$. Here, nodes "ab" and "ac" from $T^\text{odd}_S$ and $T^\text{even}_S$ was merged into the same node in $M_S$. This was too much, however, as the commonalities between the two is just "a", as is evident from $T_S$. The overmerge must therefore be partially unmerged.
First, the length of \( v \) is changed to \( d(v) \), where \( d(v) \) denotes the LCP depth of \( v \). Now, as \( v \) was erroneously overmerged, it is likely that this error has propagated further down the subtree of \( v \) because of the recursive nature of the overmerging procedure. To avoid unmerging the entire subtree, recall how \( v \) was created in the first place. The two nodes, \( m^{\text{odd}} \) and \( m^{\text{even}} \), shared the initial character on their respective parent edges, and thus needed some merging.

The assumptions were then that they were entirely identical (or the shorter of the two a prefix of the longer), because determining exact commonalities was too expensive. Now, however, it has been established that \( m^{\text{odd}} \) and \( m^{\text{even}} \) share exactly \( d(v) \) initial characters, after which they differ. This means that by inserting \( m^{\text{odd}} \) and \( m^{\text{even}} \) as children to \( v \) in a lexicographically sorted manner, when \( v \) has been modified to represent exactly this common prefix of the two, the subtree rooted in \( v \) will be correct.

**Sorting the unmerged nodes**

Finding these characters after which to sort is straightforward: Let \( T^{\text{odd}} \) and \( T^{\text{even}} \) be the descending leaf nodes from \( T^{\text{odd}} \) and \( T^{\text{even}} \) respectively, as marked on \( v \) during the overmerge. The symbols by which to sort \( m^{\text{odd}} \) and \( m^{\text{even}} \) after are \( S[T^{\text{odd}} + d(v)] \) and \( S[T^{\text{even}} + d(v)] \), i.e. the symbols in these two nodes immediately following their LCP, as represented by \( v \). These steps are illustrated in Figure 34.
Limiting the unmerging process

Because faulty overmerges propagate errors downwards in the overmerged tree, one should aim to locate and adjust erroneously overmerged nodes as far up towards the root of the tree as possible. By doing the described identifications and adjustments as part of a breadth-first traversal of the tree, one can easily filter out of the traversal the subtrees rooted in nodes in which adjustments were made, hereby limiting the number of visited nodes. When the traversal is completed and all erroneously overmerged nodes has been adjusted, a correct suffix tree has then been constructed.

Time complexity of the unmerging

As the adjustments themselves are constant-time, and no more than $O(n)$ overmerged nodes will exist, this adjustment procedure of unmerging is bounded by $O(n)$.

6.7 Choices made different

Not all parts of the description of Farach’s algorithm as laid out in [25, Ch.5.2.4] was followed directly. In certain places, we did things differently in order to make our implementation function as intended. These differing choices are outlined in the following.

Heterogeneous neighbouring node pairs

In the correction of the overmerge tree, we are comparing the depth of internal nodes in the LCP tree with the actual string length. If these are not matching, an overmerge has occurred and we adjust the nodes surrounding this faulty overmerge as previously described.

Our way differs completely from the one found in [25], as our perception and implementation of the method found in [25] led to faulty overmerged nodes not always being detected.

Our perception of the method found in [25]

The approach from [25] describes how the nodes for which an overmerged node is the LCA can be identified by considering neighbouring heterogeneous pairs of nodes in the list of leaf nodes ordered by occurrence in a depth-first search of the overmerged tree $M_S$. Each of the pair of heterogeneous neighbour nodes will have an LCA, and all these LCA nodes, when considered collectively, will correspond to all overmerged nodes as we understand it.

This list of neighbouring heterogeneous node pairs is identified in a way not clearly apparent to us in [25]. Our take on the algorithm used in [25] for identifying this list of neighbouring nodes works
on the example given in the book, and generates the list using the following approach:

We keep track of a pairing node, a node with which we will create node pairs to fill in the list. Let the first node visited be the pairing node initially. Visiting the remaining nodes in the list of leaf nodes ordered by occurrence in a depth-first search of $M_S$, we will consider the parity of each, i.e. whether they are even or odd, and compare the parity of this currently visited node to that of the current pairing node. One of two cases will apply in any such comparison: Either the visited node has a parity matching that of the pairing node, or it has the opposite parity.

In case of the nodes having opposite parity, a node pair consisting of the two is created, and the traversal continues. In the other case, i.e. their parity matches, the last seen node of opposite parity of the current pairing node is replacing the pairing node, and a pair of this new pairing node and the currently visited node is created, and the traversal continues with the new pairing node.

By following this, one will be left with a list as presented in the book when using the example given in [25], summarized here:

$4, 3, 1, 8, 5, 12, 2, 7, 11, 9, 6, 10, 13$

The list of node pairs yielded is the following

$(4, 3), (4, 1), (1, 8), (8, 5), (5, 12), (5, 2), (2, 7), (2, 11), (2, 9), (9, 6), (9, 10), (10, 13)$

Doing this, with an approach yielding the same results on the example given in the book, we found two cases in which we did not detect a crucial internal node which needed to be adjusted, resulting in an incorrect suffix tree eventually.

**Failing input example**

Using the approach from [25], if one considers the result on the list of nodes as seen for the tree in Figure 35, namely

$1, 2, 8, 5, 4, 3, 9, 6, 10, 7, 11$

One will find that the list of neighbouring heterogeneous node pairs is as follows

$(1, 2), (1, 8), (8, 5), (5, 4), (4, 3), (4, 9), (9, 6), (9, 10), (10, 7), (10, 11)$

In this example, an overmerge has occurred in the first internal node with edge "2..". To consider this node for adjustments, we have to identify a node pair with this internal node as their LCA. These are either $(2, 8)$ or $(2, 5)$, none of which are in the list of node pairs found.
Figure 35: An overmerged tree $M_S$ that needs adjusting. By using the approach described, we found a list of heterogeneous neighbouring nodes not including either $(2, 5)$ or $(2, 8)$ which are the node pairs for which the left internal node is the LCA. One of the two node pairs must be identified in order to consider it for unmerging in the algorithm.

This is due to the tree shown in Figure 35 having a structure for which we failed to identify the node pairs when using this approach. In figure 36, two structures we identified as causing this problem is illustrated. There may exist more, as solving these two structures separately only led to more subtle cases. These are chosen as to present the problem, not to claim that the problem is with these two structures only.

Our solution

By proper bookkeeping of the nodes for which an overmerged node is the LCA during the process of overmerging $T_{odd}$ and $T_{even}$, we avoid these situations. This way is more complicated to implement than the simple traversal of a list of occurring leaf nodes, however, this was the way for us in which to make sure the resulting list matched the expectations. This way, when considering the LCA of the node pairs in the list, there is exactly a one-to-one correspondence with all the overmerged nodes, as the node pairs are created in all cases of the overmerging process creating an internal node. The approach from the book may have had more node pairs referring to a single overmerged node, which causes redundant checks.

The point of the node pairs is to detect overmerged nodes in linear time. To detect all possible nodes that are overmerged, we keep track of these during the overmerging process instead of trying to infer these from a depth-first search of the resulting overmerged tree.

Every time we merge the even and odd tree, we keep track of both an even and odd leaf from two different subtrees below the merged node once the merge is complete. This ensures that the LCA of these two nodes is the one in which these were marked. By
Figure 36: $M_S$ constructions that will not be adjusted correctly using the approach given in [25] as we perceive it. Nodes labeled with an $e$ is a representant for an even node, and the same goes for $o$ representing an odd node.

doing this for all nodes created in the overmerging process, the overmerged nodes will all have a node pair by construction, as required for the detection of faulty overmerges following this in the construction algorithm.

### 6.8 Time Complexity Analysis

Every step of the algorithm, apart from the recursive call during the construction of $T_{odd}$, was argued throughout to be asymptotically bounded by $\Theta(n)$.

Because of the recursive call, the analysis of the time complexity of the subprocedure constructing $T_{odd}$ must be treated as a recurrence relation.

Let $T(n)$ denote the time it takes to produce a complete suffix tree on an input of length $n$. Then, as the construction of $T_{odd}$ requires a recursive call with an argument of size $n/2$ taking time $T(n/2)$ on top of what else happens in this step, argued to fall within $\Theta(n)$, this results in $T_{odd}$ having a construction time of

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

Therefore, to asymptotically bound the construction of the full suffix tree on input of size $n$, the following recurrence must be solved w.r.t. $T(n)$:

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

where $T(n/2)$ is the time complexity of the recursive call during the construction of $T_{odd}$, and $\Theta(n)$ encaptures the complexity of all the other subprocedures, as argued throughout.
Recurrences are solved using the Master theorem, as described in [4, p.94], repeated below for convenience.

**Theorem 4.1 (Master theorem)**

Let \( a \geq 1 \) and \( b > 1 \) be constants, let \( f(n) \) be a function, and let \( T(n) \) be defined on the nonnegative integers by the recurrence

\[
T(n) = aT(n/b) + f(n),
\]

where we interpret \( n/b \) to mean either \( \lfloor n/b \rfloor \) or \( \lceil n/b \rceil \). Then \( T(n) \) has the following asymptotic bounds:

1. If \( f(n) = O(n^{\log_b a - \epsilon}) \) for some constant \( \epsilon > 0 \), then \( T(n) = \Theta(n^{\log_b a}) \)
2. If \( f(n) = \Theta(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a} \log n) \)
3. If \( f(n) = \Omega(n^{\log_b a + \epsilon}) \) for some constant \( \epsilon > 0 \), and if \( af(n/b) \leq cf(n) \) for some constant \( c < 1 \) and all sufficiently large \( n \), then \( T(n) = \Theta(f(n)) \). □

Looking at relation (6) again with the Master theorem in mind, one will find values \( a = 1 \), \( b = 2 \) and \( f(n) = n \). With these values, case 3 will apply

\[
f(n) = \Theta(n) \Rightarrow f(n) = \Omega(n) = \Omega(n^{\log_2 1 + 1}),
\]

when \( \epsilon = 1 \) is chosen. Further,

\[
af(n/b) = 1 \cdot n/2 = 1/2 \cdot n = 1/2 \cdot f(n) = cf(n),
\]

for \( c = 1/2 < 1 \). Thus, the asymptotic bound of the recurrence must be \( T(n) = \Theta(n) \).
Part II

PRACTICAL EXPERIMENTS
It would seem the performances of the algorithms have already been covered throughout by discussion of their worst-case running times. However, these are just that: the theoretical worst-case performance of the algorithm. In practice, this may not be the function best describing the running time as a function of input length. As an example, the naive algorithm had a worst-case running time of $O(n^2)$. This was because there was a risk of scanning the whole input string in order to place the second suffix, if it and the first agreed on all but the very last character of the second suffix. Although, had they differed on the very first character, and if this was the general trend, then one might have observed near-linear running time of the naive algorithm, as a mismatch could be identified in a single operation for each suffix inserted.

What is presented in the following is a portrait of the relative running times in practice of our implementations of the three presented algorithms when given various types of input.

The efficiency of implementations of an algorithm can be measured on a variety of metrics of potential interest, depending on context of usage. In this section, however, focus will be on the exclusively on time spent constructing the trees in the conducted experiments, which is thus what is meant by performance from here onwards. Emphasis must be put on the presented performances being nothing but that of our particular implementations of the algorithms, not the algorithms themselves, and that these implementations are a first take, likely with plenty of room for optimizations. Thus, the results of the following experiments will be a reflection not only of the algorithms, but also of our specific implementations.

### 7.1 Environment

All experiments will be computed using Python version 3.6.1 on OS X El Capitan version 10.11.6 on a computer with a 2.4 GHz Intel Core i5 processor and 8 GB ram.

The computer is a laptop used for daily tasks, and thus will have other background processes running alongside the experiments. An effort has been made to close these other running processes when experiments were being computed. The computer was plugged in as to not let battery drain potentially cause performance fluctuations.
7.2 Implementations and How to Use

Our implementation of the algorithm can be found at https://github.com/skodborg/farach-suffix-tree. Each of the mentioned subprocedures in the description of Farach’s algorithm has an associated function, easily identified by its name, in the file farach.py. The implementation of radix sort used is the one found at [7], slightly modified to suit our needs.

How to use

The three algorithms are implemented in the files farach.py, mcreight.py and naive.py, respectively. Each file can be run using Python 3.

If the flag -f is given, the first argument will be handled as being an input file, otherwise the argument will be used as input. An output file can be specified by using the flag -s. If no file is specified, the tree will be printed in the console.

```bash
> python3 farach.py "mississippi"
> python3 farach.py inputfile.txt -f
> python3 farach.py inputfile.txt -f -s outputfile.txt
```

Figure 37: Variations of running Farach’s. The naive algorithm and McCreight’s can be run using identical arguments.

7.3 Verification of Implementation Correctness

Performance is meaningless if the implementation does not function as intended. Thus, some means of verifying that the trees output by each algorithm are in fact correct suffix trees of the given input is necessary, where a tree is considered being a correct suffix tree if it is in accordance with the definition presented in 2.6.

The implementation of the naive algorithm will partly function as a verification tool for the implementations of McCreight’s and Farach’s. By careful implementation of the naive algorithm, manual testing on several example input strings with known corresponding suffix trees, and due to the simple nature of the algorithm, we are confident that the implementation hereof is performing as intended, outputting correct suffix trees. Note that this is not a guarantee of correctness but rather an establishment of credibility of the implementation.

Further, we implemented a function which, when given a string and a tree, will traverse the tree, testing whether it is in accordance with the definition presented in Section 2.6 of suffix trees over a given string. This function was then given the output of the naive algorithm on several randomly generated inputs in an automated looping pro-
procedure, and they seemed to agree on what a suffix tree of a given string looks like.

Convinced that the naive algorithm was implemented correctly, and with the function verifying that a tree abide by the definition of a suffix tree, verification of correctness of the more complex implementations of McCreight’s and Farach’s algorithms respectively was done as such: 10,000 strings was created, each being of length 1000 drawn uniformly at random from an alphabet of size 10. For each of these strings, a suffix tree was constructed by each of the algorithms, yielding three trees. These trees were then traversed and compared to that of the others, ensuring that they were in fact identical trees. Lastly, one of these three identical trees were traversed by the function testing whether the tree was in accordance with the definition of a suffix tree over that particular string. When all 10,000 tests completed successfully, we were confident in the correctness of the implementation of the remaining two algorithms, namely McCreight’s and Farach’s as well. Note again, that correctness is still not guaranteed for the implementations, merely being likely true to the extent one approves of the testing done.

Choice of test data

The choice of random data to base the correctness tests upon is as good as any for testing the correctness of the implementation. An exhaustive test of all possible strings is obviously infeasible. The randomness of the input will, however, provide versatility in input strings as well as result in occurrences of repeated substrings, which will result in some non-trivial structure between internal nodes in the constructed tree. The existence and expected lengths of these repeated substrings occurring in uniformly randomly drawn strings is, as described in [14, 15, 24], \( O(n \log |\Sigma| n) \).

Outputting correct results is not enough to conclude correct implementation. The algorithms should at least show a performance within the alleged theoretical worst-case running times as well. Otherwise the implementation of the algorithm is considered faulty, whether by accident or as a result of a misunderstanding of some aspect of the algorithm at hand. Any comparison of faulty implementations of the algorithms will be untrustworthy.

Both the worst- and the best-case inputs for the naive algorithm are being tested to verify the implementation and performance hereof. As McCreight’s and Farach’s are both linear time algorithms, the worst- and best-case inputs for these two respectively will only show as a change of a constant factor between the two types of inputs. Thus, an initial test on identical data as used to verify the correctness of the output, however growing in size each iteration, will be used to give an initial impression of their relative performances while also showing the running times of McCreight’s and Farach’s to be no worse than
linear to somewhat convince of the correctness of this aspect of their implementations, too. This is shown in Section 8.1 after the following brief summary of all experiments performed.

7.4 DATA CHOICES

The following experiments are divided into artificial data and real data. This is because aspects of the data assumed to have a direct influence on the construction time of the algorithms can better be controlled in an artificial data environment, whereas the real data will give an impression of to which extend these aspects hold for plausible input types given to the construction algorithms. The real data experiments will potentially further reveal if more aspects than those suspected of influencing the construction time exists. If the performance on real data is not in line with the expectations, as based on knowledge of the degree to which the aspects was present in the real data, then other factors must influence the construction time as well. These could include, but is not limited to, inherent structures in the real data, such as that of DNA encodings which is a data representation of a system with underlying structures not likely present in the uniformly chosen random data.

The amount of points in the plots is chosen rather arbitrarily by us, based on samplings of construction times given input of varying lengths and estimates of the time it would take to produce an interesting number of points within a reasonable time frame being based hereupon.

Unless otherwise stated, all plot points mark the mean of five samples of construction times of a given input type, with the standard deviation marked by a hatched area. All timings were done using \texttt{time()} from the \texttt{time} module in Python 3.6.1, more on which can be found in [22].

7.4.1 Artificial data

With these experiments, we seek to determine to what degree various aspects of the data will influence each of the algorithms in order to understand and compare performances of them.

The following is a brief overview of the experiments conducted on artificially constructed data found in this chapter.

- \textit{Uniformly randomly drawn data strings} - These were drawn from an alphabet of size 10, increasing the length of the input string. This is to gain an initial impression of their relative performances, as well as indicating whether the two linear-time algorithms do in fact perform in linear time as intended.
• **All different characters** - This is the best-case input for the naive algorithm, as it minimizes the number of character comparisons necessary in each suffix insertion. The input is given to all three algorithms, to indicate their relative performances.

• **All identical characters** - Contrary to the above, this is the worst-case input for the naive implementation due to the maximization of necessary character comparisons. This is chosen as to contrast the prior best-case input for the naive algorithm, and as such form an impression of the two linear-time algorithms in relation to the naive algorithm in each case of it performing at its best and its worst.

• **Periodic strings with varying periods** - These experiments are undergone as to controllably vary the sum of the commonalities of all suffixes of a string, which essentially is what is represented by a suffix tree, and thus expected to affect the construction time hereof.

• **Fixed alphabet sizes** - This is assumingly where a difference in behaviour of Farach’s and McCreight’s algorithms is to be seen, due to the difference in dependency hereon in the two algorithms. The intend of this experiment is to show the presence of this dependence difference, and give an impression of as to which extent it influences performance.

• **Relative alphabet sizes** - By enforcing a certain percentage of the length of each string to consist of unique characters, experiments are conducted in which the ratio between length and uniqueness of an input is fixed. This is a follow-up experiment, intended to further explore and compare the relation to alphabet size of each of McCreight’s and Farach’s algorithm.

### 7.4.2 Real data

One thing is artificially constructed data where aspects of the data can be controlled to investigate their influence on the time spent by the suffix tree construction algorithms, another is to which extend these aspects are realistic, if at all.

As an example, consider the sum of the LCPs of all suffixes. This sum encapsures the total amount of what could be described as the commonality of all suffixes of the input string. This is essentially what a suffix tree holds information about in all its layers of internal nodes between the root node and leaf nodes. Imagining this sum to vary with different types of data is not hard, and seeings as it affects the tree as evident from Section 8.4, it is interesting in which order of magnitude one should expect this particular aspect of data being present in real data on which suffix trees might be constructed.
The following is an overview of experiments conducted on real data, with the intention of indicating to what degree each of the artificial experiments is relevant in possible real cases of suffix tree construction.

- **DNA data** - Two DNA datasets were acquired from The National Center for Biotechnology Information [8], namely the *BRCA1* gene and the *Human Chromosome 2*.

- **English text** - In this experiment, the complete novel of *Moby Dick* was acquired from Project Gutenberg [19], chosen as to represent written English text.
ARTIFICIAL DATA

8.1 INITIAL IMPRESSION

In order for us to get an initial impression of the relative performances of the three algorithms, the time spent by each to construct a suffix tree of a string, drawn uniformly at random from an alphabet of size 10, is measured. As with the correctness verification, the choice of data drawn uniformly at random is as good as any for the purpose of getting an initial impression, as well as seeing a linear trend in construction times for the two linear-time algorithm implementations, indicating that the implementations do in fact function as intended.

Experiment details

For each plot point in the graph, five different strings of a given length was constructed by drawing characters from an alphabet of size 10 uniformly at random. Each of these five strings were then given as input to each of the three algorithms, and the construction of the corresponding suffix tree was then timed. The length of the given input string is incremented by 100,000 for each following plot point.

Observations

Looking at the graph in Figure 38, it seems clear that all three algorithms is running in linear time, indicating that Farach’s and McCreight’s have been implemented as intended, with running times in practice showing to match their theoretical time complexities. At first glance, our implementations of McCreight’s and the naive algorithm looks to be following each other closely, with McCreight’s being slightly slower than the naive, while the implementation of Farach’s is a lot slower than both of these two.

Discussion

That the naive algorithm performs in linear time, despite its worst-case theoretical time complexity, is expected. This theoretical worst-case is a maximization of the number of characters that must be scanned in order to insert each suffix. For uniformly randomly drawn strings, the LCP of two suffixes of the string is the same as a repeated substring in the string, of which we know the expected length to be $O(\log |\Sigma| n)$ according to [14, 15, 24], i.e. dependent on both the size
of the alphabet and the length of the string. In this particular experiment, this expectation amounts to $\log_{10} 650,000 \approx 5.81$ at its highest, which is thus the expected length of each LCP scanned per insertion of a suffix in the tree. That this shows to be faster than the overhead in the implementation of McCreight’s is no surprise, and McCreight’s tricks being able to fastscan past 5 characters simply is not justifiable by the overhead present in our implementation.

As expected, the performance of Farach’s is linear with input length on this type of data, but clearly slower than the other two algorithms. In Farach’s, no matter the data, it will always call recursively on the input string, creating suffix trees of all even and odd suffixes, respectively, and proceeding to merge these together. This results in a series of steps that are required in every creation of a suffix tree, even though the tree might be simple with few internal nodes representing short strings, which is reflected in the much larger slope of the plot line of construction times of Farach’s.

### 8.2 All Identical Characters

Worst-case for the naive is a string of all identical characters. This will maximize the sum of the LCPs of all suffixes of the string, which essentially is the amount of work the naive algorithm has to do as it scans along the longest LCP of a suffix and all previously inserted suf-
fixes. The growth of this sum as the length increases is within $O(n^2)$. Beyond scanning this sum of characters, it must create and insert at most $2n$ nodes in the tree, each in time $O(1)$, as well as locate the single downward edge to walk along from the root, which is performed in time $O(1)$, making all other work being within $O(n^2)$, which is what we expect the performance of the naive algorithm implementation to show.

**Experiment details**

Strings were constructed consisting of only one character, repeated an increasing number of times, starting at 1000 occurrences with increments of 1000 occurrences more for each successive plot point.

The naive implementation is in line with its worst-case time complexity

This experiment shows the expected $O(n^2)$ running time for the naive algorithm, where the running time is plotted as a function of the length of the input given, followed by a graph of the running time divided by the expected running time function, namely $n^2$ as a function of $n$. If the assumption is correct, this second graph should show exactly the constant of $O(n^2)$ on the y-axis if there is in fact a constant ratio between the time it takes the naive algorithm to process input of length $n$ and the function $n^2$ for growing values of $n$. This seems to be the case, however the graph does show a declining trend. When considering the zoom level of the graph in relation to the actual running times, e.g. a difference of roughly $0.1 \cdot 10^{-7}$ between inputs of length 1000 and 30,000, where the latter takes close to 140 seconds to construct, we feel this declining trend is neglectable, and that the naive algorithm does in fact show behaviour within $O(n^2)$.

**Evaluation of McCreight’s**

McCreight’s algorithm seems to perform really well on this input too, which is to be expected. The input allows for full usage of the `fastscan` strategy for locating the position of the next insertion point after slowscanning and inserting the second suffix. From the point of inserting the second suffix and onwards, the `head` node will always be an internal node different from the root node for this particular type of input, allowing for fastscanning for every subsequent suffix insertion. With only one symbol in the alphabet, locating the downward edge from the parent of `head` is possible in constant time, and the searched position will always be located between the root node and `head` for this type of input, allowing for $O(1)$-time insertions of every suffix onwards. This essentially makes McCreight’s perform linear in input size with a very low constant.
Figure 39: The relative performances of the algorithms on input strings of all identical characters, the worst-case input for the naive algorithm. As expected, McCreight’s is able to fully utilize its fastscan strategy, thus being fast. Farach’s unexpectedly reported an error, which is elaborated upon in Section 8.2. The naive shows performance indicative of being in line with its expected worst-case of $O(n^2)$. This is further established by the bottom graph, where the running time of the naive is divided by $n^2$, showing a constant ratio. The downward trend on this we believe is neglectable, given the values on the y-axis being so small relative to the construction times themselves.
Evaluation of Farach’s

As for Farach’s, unfortunately we got a maximum recursion depth exceeded error as a result of exceeding the number of recursive calls allowed by the python interpreter. This happened during the construction of $M_S$ whenever two nodes were overmerged. Recall that overmerging happened when two nodes from $T_{odd}$ and $T_{even}$, respectively, were considered, and one of three cases was applied to complete the overmerge. Two of these cases results in a recursive call, namely those for which the initial character on each of the parent edges of the two nodes had to match. As there is only one character in the alphabet in this experiment, this holds in all cases, including the overmerging in all the following recursive calls down the subtrees. This ultimately results in a number of nested recursive calls roughly proportional to the number of internal nodes, which, for this tree, is approximately $n/2 - 1$ as the tree is a complete binary tree, disregarding the root node.

This gives rise to considerations regarding the recursive solution of the algorithm and the implementation, which will be further discussed in Section 10

8.3 All different characters

The best-case input for the naive is a string of all different characters. As previously mentioned, the naive algorithm is roughly doing work proportional to the sum of the LCPs of all pairs of suffixes of the given string. As all characters are unique in the string in this best-case, no suffix will have an LCP of positive length, and thus a minimum of characters must be compared for each suffix insertion. These character comparisons amount to exactly those needed to determine that no other existing edge shares the initial character of the suffix being inserted at any given point.

Experiment details

The input in this experiment were constructed as being $n$ unique characters for inputs of length $n$, where these input lengths started at 10,000 and incremented by 10,000 for each successive plot point.

Evaluation and comparison of McCreight’s and the naive

McCreight’s performs roughly identically to the naive. This is expected, as every clever idea utilized in McCreight’s algorithm is nullified by the data type. No internal nodes besides the root node will exist in the suffix tree, thus no suffix links other than the root pointing to itself will exist which in turn means no shortcuts will be taken. Also, as no suffix links can be followed, no guarantees of the existence of strings in the tree can be made which in turn means that fastscan
Figure 40: The relative performances of the algorithms when given input strings consisting of all unique characters. This is the best case input for the naive algorithm, which is also the fastest of the three here. McCreight’s performs slower than the naive as it is a more complex implementation, and the input does not allow for any use of the two tricks characterising McCreight’s algorithm. While Farach’s is still slower than the others, it is faster than it was when given a string with multiple occurrences of each character. This is expected, as the suffix tree in this case will contain no internal nodes apart from the root, whereas it will with multiple occurrences of identical characters. Not having to create and insert these nodes saves time.
will never be used. However, *slowscan* will only ever have to compare the first character of a suffix with all previously inserted suffixes to determine no matches, much in line with the naive algorithm. This effectively reduces McCreight’s to function very much alike the naive, with some added overhead in the implementation, which we think is what the graph shows in the small difference we are seeing when trees are large enough.

*Evaluation of Farach’s*

As for Farach’s, while clearly being slower than the other two, it is still faster than it was on the uniformly randomly drawn input, hinting at diversity in data reflecting positively in total time spent on construction. This is a general trend, however, for suffix tree construction algorithms, as their inherent nature is reflecting commonalities in suffixes. With close to none of such commonalities, every suffix tree should be simple. In the particular case of Farach’s algorithm, as no internal nodes other than the root node exists, the overmerging of trees $T_{odd}$ and $T_{even}$ is reduced to a simple merging procedure, with no cases of handling identical characters occurring. This will further result in the construction of the LCP tree and the adjustments of overmerges being completely skipped.

8.4 **Periodic strings**

Commonalities of substrings is what essentially makes up the complexity of constructing a suffix tree, making them an interesting subject for experimentation. The random aspect of uniformly randomly drawn strings does not allow for full control over these commonalities beyond setting up a model for calculations of expected values for the sum of all LCPs of all suffixes of a string drawn in this fashion. However, periodic strings do, as will be introduced here.

*Definition and example of a periodic string*

Let $u$ be some string of length $p$, and let $u'$ be some possibly empty proper prefix of $u$. Let $r \in \mathbb{N}$. We define a periodic string $u'u'$ as being the string $u$ concatenated with itself $r$ times, followed by a final concatenation with $u'$, resulting in a string consisting of $r$ occurrences of adjacent identical substrings of length $p$, apart from the possible abrupt ending of a repetition at the end of the string. Note that for a given string, there can be multiple valid ways of describing it by this format $u'u'$. In particular, when talking about a string being periodic, it has a way of being partitioned into this format $u'u'$, with the minimal value of $p$, i.e. the minimal length of $u$, among these valid partitions being referred to as the *period* of the string. Not all
Strings are periodic, i.e. able to be partitioned in this way, and those that are not are called primitive strings.

An example of a periodic string is "abaabaab" having $u = "aba"$ with $u' = "ab"$ and a period of $p = 3$. As $u'$ is allowed to be empty, $u = "abaabaab"$ with $u' = \epsilon$ and $p = 8$ is another valid partitioning of the string into this format, however $p = 8$ is not minimal here due to the existence of the valid partitioning having $p = 3$ with $u = "aba"$. Also, for the choice $p = 2$ with $u = "ab"$, it follows then that the only possible value of $r$ is $r = 1$, which would then require $u' = "aabaab"$ to cover the entire string using this format $u'u'$. This is an invalid partitioning as $u' = "aabaab"$ is not a proper prefix of $u = "ab"$. The same applies for the other choices of $p < 3$, making $p = 3$ the period of the string.

Remarks on the particular periodic strings used

Important here is the fact that the periods were chosen to consist of all unique characters. The structure of the repeated string influences the number of internal nodes contained in the resulting tree, which is known to affect overall construction time. The chosen format of the repeated string allows for quick determining of the growth of the number of internal nodes as a result of increasing the input length, as an internal node will be created for each suffix inserted past the first
p suffixes, where p is the length of the period repeated in the input string, resulting in \( n - p \) internal nodes in a tree created in this way. This allows to more easily determine the influence of node growth on the construction time, and lets for a more qualified explanation of the running time observed for the specific algorithm when given periodic inputs.

On this particular choice of input data, it being repetitions of strings consisting of unique characters, the naive algorithm will use the first \( p \) iterations determining that no previous suffix shares any prefix with it, and it will eventually be inserted as a direct descendant from the root in time \( O(\log |p|) \). However, once these \( p \) iterations has been processed, the \( p + 1 \)'th suffix will share its full length with the first suffix due to the strings being periodic. All successive suffixes \( i \) will likewise share their full length with the previously processed and inserted suffix \( i - p \), thus requiring the naive algorithm to scan the full length, resulting in \( n - p \) iterations of insertions taking \( O(n \log |p|) \) each. As \( p \) is increased, the number of expensive insertions taking \( O(n \log |p|) \) will decrease for two inputs of identical lengths. Thus, the longer period will be faster due to the increased number of fast operations. Every slow operation will also result in the insertion of an internal node, which was estimated to be \( n - p \), thus decreasing proportionately to an increase in \( p \). This results in a running time of \( O(n^2) \) for these particular choices of data.

**Comparison of plot lines in Figure 41**

The worst-case performance of the naive algorithm is \( O(n^2) \), which is in line with what is seen in Figure 41. Here, the plot line for \( p = 1000 \) is slightly below those corresponding to the choices of \( p \) being 1, 5, 10, and the plot line for \( p = 5000 \) is even further below that of \( p = 1000 \). The difference between choices 1, 5, 10 of \( p \) boils down to at most 10 slow insertions with a 10 node difference in their respective trees, and as such, we believe the differences shown in the graph to be noise as a result of other processes running concurrently with the experiments on the computer.

Note, that the experiment with \( p = 1 \) is in fact equal to, yet worded differently than what was previously referred to as the worst-case input for the naive algorithm, of which the results of an experiment hereon was plotted in Figure 39. This is because this worst-case input is essentially periodicity taken to the extreme by maximizing the number of repetitions possible, achieved by choosing the minimum sized period length, i.e. \( p = 1 \).

This has the effect that commonalities between suffixes, i.e. the sum of LCPs of all suffixes, is being maximized, hence an excerpt of the data from this previous experiment, matching the length of the other periodic strings in this new experiment, is re-plotted in this graph.
As this worst-case data was produced at a completely different time than the rest of the data for the other periodic strings, and seeing as all that should differ between these graphs was determined to be at most the time it takes to locate and insert 10 nodes, this further backs up the claim that the difference in background processes at the two points in time has caused these indistinguishable graphs for periods 1, 5 and 10.

*On singling out the naive algorithm*

With the naive algorithm clearly performing worse eventually on all variations of lengths of the period tried than the worst-case for both the linear algorithms, and with the speed of the naive and the two linear-time construction algorithms only being comparable while the string consists of all unique characters until the full period is covered, in which case we already have an impression of their relative performances as shown in Figure 40, a comparison of the linear-time construction algorithms and the naive algorithm seems redundant. This leaves a comparison of the performance of the two linear-time construction algorithms on this type of input to be made, which will be found in the following.

*Evaluation of McCreight’s*

As for McCreight’s, the first $p$ suffixes is inserted in time $O(\log |p|)$, the time it takes to determine that no other previously inserted suffix shares a prefix with the current suffix, as is the case for these periods consisting of all unique characters. The $p+1$th suffix must be localized by slowscanning in time $O(n \log |p|)$, but the remaining suffixes will all be inserted by fastscanning because the head will be different from the root in all successive iterations from this point. This means the remaining $n - p - 1$ suffixes will each be inserted in time $O(\log |p|)$, the time it takes to locate the edge to fastscan along and insert at, with fastscanning and insertion each being $O(1)$. Fastscanning being $O(1)$ is because the parent of head is always the root in this particular case, as any new internal node will be inserted between root and the previous head node.

This results in a worst-case performance of $O(n \log |p|)$ for McCreight’s. However, the difference in construction times seen between plot lines for periods $10 \cdot 10^3$ and $25 \cdot 10^3$ for McCreight’s, as well as Farach’s, algorithm cannot be justified by the factor of $\log(25 \cdot 10^3)$ and $\log(10 \cdot 10^3)$ in the worst-case performance. The difference between these two $\log(25 \cdot 10^3)$ and $\log(10 \cdot 10^3)$ is roughly 1.3, which is not enough to describe the differences seen in Figure 42 for both Farach’s and McCreight’s for these two plot lines.

This difference observed is, however, likely directly related to the number of internal nodes in the trees. The number of these was ar-
8.4 Periodic Strings

Figure 42: The performance of Farach’s and McCreight’s when given periodic strings with varying periods as input. As expected, Farach’s plot lines has a higher slope than the corresponding plot line for McCreight’s, assumingly due to overhead in implementations. Two of the plot lines of Farach’s are cut off at input lengths of approx. 20,000 and 32,000, respectively. This is due to choices in the implementation, and is further elaborated upon in Section 8.4. The upward bend at input length 25,000 on period length $25 \cdot 10^3$ as seen in the plot lines of both algorithms is due to the experiment, as this is the point where repeated occurrences of symbols starts taking place. The plot lines for each algorithm seem to have a decreasing slope with an increase in period. This is similar behaviour as seen with the naive algorithm, and is likely due to the suffix trees having fewer internal nodes with higher periods for these periodic strings.
guessed to be $n - p$, decreasing as $p$ increases. That this is the cause of the performance difference in our implementations of the algorithms further shows in the graphs of the two algorithms particularly around input length $25,000$, where both graphs for period length $25 \cdot 10^3$ seem to start bending upwards. This is the point at which a new internal node is introduced per character above the initial $25,000$ characters in the input, indicating that the change in slope of the graph just around this point is directly reflectant of the time spent by our implementations to create the internal nodes.

**Evaluation of Farach’s**

In Farach, a *maximum recursion depth exceeded* error was reported for two of the input types tried, namely period lengths $10$ and $25 \cdot 10^3$ once input lengths passed lengths $9000$ and $32,000$, respectively. Each is a result of the two uses of recursion used in the implementation of the algorithm, hinting at which structures in the data our implementation of Farach’s struggles with.

For period lengths of $25,000$, the choice of recursion as a solution to annotating the overmerged nodes with their depth in the LCP tree, as laid out in Figure 32, is the problem. This is specifically a result of having $25,000$ children from all internal nodes, and each of these, in case they are internal nodes, will have a suffix link pointing to its right-adjacent sibling, forcing the recursive function in Figure 32 to follow all these suffix links in a recursive call due to the choice of running this recursive function as a left-to-right breadth-first traversal. The problem lies in the recursion, not the direction taken in the traversal. If the traversal was instead right-to-left, then periodic strings consisting of periods of characters of descending order in the alphabet would cause the same forward-pointing suffix links due to the lexicographic sorting of the tree.

For suitably short period lengths, such as $10$ was shown to be, we get another *maximum recursion depth exceeded* error in the same piece of code that was troublesome during the experiment with the worst-case input for the naive algorithm consisting of all identical characters. Unfortunately, due to time constraints, we were not able to pinpoint the exact aspect of periodicity and its relation to the produced subtrees causing the recursion in the overmerge procedure to exceed its limits. However, we are not surprised to find that the recursion used is causing errors when the input grows. A possible solution to both cases of recursion in the implementation will be discussed in Section 10.

**Trend of plot lines common to both algorithms**

When looking at the plot lines for each algorithm, their slopes seem to decrease with an increase in period for these particular periodic strings, causing them to fan out over an angle when considered col-
lectively. This is a tendency too observed for the naive algorithm, and is likely based on the suffix trees for these periodic strings having fewer internal nodes the higher the periods.

Further backing up this idea is the previous results seen when comparing Figures 38 and 40. Looking at input length 300,000, for instance, in both of these graphs, Farach’s constructed a suffix tree in \( \sim 60 \) seconds on the string drawn from an alphabet of size \( |\Sigma| = 10 \), whereas in the best case of the naive with \( |\Sigma| = n \), i.e. the suffix tree with no internal nodes besides the root, a tree of length 300,000 was constructed by Farach’s in just above half this time, roughly \( \sim 34 \) seconds.

This same result is to be seen when considering McCreight’s plot lines in these two figures for this input length. McCreight’s constructs a suffix tree in time \( \sim 20 \) seconds when drawing a string uniformly at random from an alphabet of size \( |\Sigma| = 10 \), whereas a suffix tree with no internal nodes over an input string of the same length is constructed by McCreight’s in approximately \( \sim 4 \) seconds. Be wary that there is of course other differences between the input strings given to the algorithms in the experiments behind these two compared graphs, however the same tendency is in effect here, pointing towards more nodes being the cause of the decreasing slopes.

8.5 Varying alphabet sizes

In the description of the theoretical running time complexities of McCreight’s and Farach’s algorithms, it became evident that the difference between the two was the presence of a dependency of the size of the alphabet in McCreight’s, whereas Farach’s was claimed to be independent of this alphabet size. To what degree this alphabet dependence impedes the performance of McCreight’s, and when, if ever, this dependence will let Farach’s perform better than McCreight’s will be discussed in the following.

McCreight’s is essentially the naive algorithm with two added tricks to speed up its performance. As these tricks are not the source of the alphabet size dependance of McCreight’s, rather this lies with the locating of the downward edge to search along, which is common to both McCreight’s and the naive algorithm, we see no point in including the naive algorithm in these experiments, as it should show similar, albeit slower, construction time patterns as those seen by McCreight’s.

8.5.1 Fixed alphabet sizes

The initial experiments with various alphabet sizes are performed with a selection of fixed alphabet sizes from which the input strings are constructed.
Figure 43: The performances of McCreight’s and Farach’s as the size of the alphabet is varied. While Farach’s seems largely unaffected, as is expected due to its alleged alphabet independence, McCreight’s seems to vary more. The seemingly non-linear behaviour is a consequence of the shape of the experiment itself, and is further elaborated upon and discussed in Section 8.5.

**Experiment details**

The experiment was performed as such: Six fixed alphabet sizes was decided, from which strings of increasing length was drawn uniformly at random.

The two alphabet sizes $8 \cdot 10^4$ and $1 \cdot 10^5$ were included subsequently to plotting the graphs with only the four alphabets of sizes powers of two, and was included as to further indicate the observed and discussed tendencies in the following.

**Initial impressions of Farach’s performance**

Looking at Figure 43, it is clear that not much difference is to be seen in the construction times of Farach’s as the alphabet size is varied. What little variation there is between the slopes of the green lines we believe is a result of an increased number of internal nodes in the constructed suffix trees. This claim is backed by the plots shown in Figure 45, where specifically on the y-axes seen on the plots as the alphabet size is increased, the values generally increases with the alphabet, meaning that with more unique characters follows more internal nodes in this case.

As an example, consider in Figure 45 the plots corresponding to $|\Sigma| = 2^{14}$ and $2^{17}$, specifically for input lengths of $101 \cdot 10^3$. Neither the number of nodes or the number of unique characters contained in the string surpasses 20,000 for input strings of this length for $|\Sigma| = 2^{14}$, but looking at the same length input strings for $|\Sigma| = 2^{17}$, we see...
more than 120,000 of both unique characters and internal nodes. This larger amount of internal nodes clearly takes longer to create and insert, which is common to all of the three algorithms, the naive, McCreight’s and Farach’s. The increasing number of nodes is likely the result of the increased number of unique characters drawn, which has the effect that common prefixes of suffixes will be shorter in general, as the expected length of each LCP of two suffixes of a string drawn in this fashion is $O\left(\log|\Sigma| n\right)$ [14, 15, 24], making the expected LCP shorter as $|\Sigma|$ is increased.

The initial rise of plot lines for McCreight’s

As for McCreight’s, the graphs differ more, telling us that changes in alphabet size affects the running time of the construction using this algorithm noticeably. Common to all the McCreight plot lines are that they do not look to be linear at first glance. This, however, we believe is a product of the shape of the experiment, concretely once again a consequence of drawing uniformly at random from an increasing pool of possible symbols as discussed previously. To support this claim, consider the rough shape common to all the McCreight plot lines. They have a steep rise initially, as they slowly flatten out to having a more moderate, positive slope as the input string length is increasing. This matches the evolution of the red bars in all of the plots in Figure 45 and is in line with the intuition that more internal nodes means longer construction times.
Figure 45: The red bars show the medians of the numbers of unique symbols occurring in the strings drawn for each input length for each of the six alphabet sizes, with the standard deviation marked by a barely visible black vertical line atop each bar. Similarly, the green bars show the mean of the number of internal nodes in the resulting suffix trees. Note in particular, that the number of unique symbols and the number of internal nodes are both medians over five iterations, and thus not necessarily from the same exact iteration.
The eventual flattening of McCreight’s plot lines

Looking at the graphs of McCreight’s for alphabet sizes $|\Sigma| = 2^{14}$ and $2^{15}$, the initial rise, if any, is hidden in the blue box, which will be further studied later. What is interesting about these graphs in particular, however, is that they show the expected linearity of McCreight’s algorithm, once the input length is large enough relative to the alphabet size. This is further evident from Figure 45, where the green and the red bars quickly even out to roughly the same height, as opposed to the remaining four plots in this figure. It is the case for both, but more clear from the plot for $|\Sigma| = 2^{14}$ than for $|\Sigma| = 2^{15}$ in Figure 45, that the number of internal nodes will eventually surpass the number of unique characters found in the string. As the number of unique characters in the string is obviously limited by the size of the alphabet, convergence hereto is expected, but the number of initial nodes will keep increasing with the input length. This is the point in which we expect McCreight’s to show linear behaviour for a given alphabet size, when the unique symbols occurring converges to a fixed number. Unfortunately, the chosen alphabet sizes and the input lengths we were able to test on within reasonable time did not allow for convergence of all graphs, but they all come close.

Had we continued the experiments and kept incrementing the input lengths, the linear behaviour of graphs for alphabet sizes $|\Sigma| = 2^{14}$ and $2^{15}$ for McCreight’s is the linear behaviour we expect the other four graphs to also have eventually. As evident from Figure 46, the plot lines of McCreight’s and Farach’s is argued to cross eventually somewhere beyond the graph showed in Figure 43.

Vertical distances between plot lines

The vertical difference between the two graphs of McCreight’s for alphabet sizes $|\Sigma| = 2^{14}$ and $2^{15}$ is the result of the dependence on the alphabet of the algorithm. It is not difficult to imagine that this vertical distance being larger for the larger increases in alphabet size is in fact reasonable, as it is tied to the initial rise of the graphs which are dependent on exactly the alphabet size as argued previously.

An example of this coherence and a possible quantification hereof, consider the McCreight plot lines for alphabet sizes $2^{14}$, $2^{15}$, $2^{16}$ and $2^{17}$ at input length 800,000. The vertical distance between the graphs for alphabet sizes $2^{15}$ and $2^{16}$ seems to be roughly four times the distance between the graphs for $2^{14}$ and $2^{15}$. Same goes for $2^{17}$ and $2^{16}$ being seemingly four times the distance between $2^{15}$ and $2^{16}$, further indicating a systematic alphabet dependence in McCreight’s algorithm. The choice of input length 800,000 was made to be far enough towards the end of the graph to allow for the graphs to have converged enough towards their alleged linear behaviour, so that little to no influence of the initial rise of the graphs will influence the
following claim, while also simply being easily identifiable by the reader.

Note that this is nothing but another claim of the alleged relation between running times and alphabet sizes in relation to one another, and the factor of four is an estimate, and is strictly tied to our particular implementations and experiments.

McCreight’s plot lines eventually crosses under those of Farach’s

The McCreight plot lines seem to have a smaller slope than those of Farach’s, once they have flattened out to their linear behaviour. If this is true, and alphabet size does not affect the slope of the lines of McCreight’s more than they do those of Farach’s, then eventually the lines of McCreight’s and Farach’s will cross again for large enough inputs. This happens in Figure 43 for alphabet sizes $|\Sigma| = 2^{16}$ and $|\Sigma| = 8 \cdot 10^4$ at around input lengths in ranges $400,000 - 500,000$ and $700,000 - 900,000$, respectively.

To support the previous claim of them eventually crossing, we decided to test construction times of the two algorithms on alphabet size $|\Sigma| = 2^{17}$ for input lengths well beyond the graph. Concretely, both McCreight’s and Farach’s were given an input string of length $2.7 \cdot 10^6$, drawn similarly from an alphabet of size $2^{17}$, and the mean and standard deviation over 10 iterations of this is plotted in Figure 46. From this it seems clear that McCreight’s will eventually be
faster than Farach’s for large enough inputs, due to the convergence of the value of the slope of McCreight’s to one of smaller value than that of the plot lines of Farach’s for similar alphabet sizes.

With this in mind, it is in particular interesting for what ratio of alphabet size and input length Farach’s is faster than McCreight’s and vice versa. We will return to this question later, in Section 8.5.2 more specifically.

**Zoom-in on Figure 43**

Looking back at Figure 43, it is hard to see exactly when the graphs overlap, and how the alphabet dependence pans out for small inputs. Figure 44 is a repetition of the experiments from Figure 43, but with a smaller initial length, along with smaller increments.

Initially, for input lengths of less than approximately 80,000, our implementation of McCreight’s is faster than that of Farach’s. This is likely due to the larger overhead in the implementation of Farach’s, as has been the general trend in previous experiments as well.

The turning point after which Farach’s looks to be faster than McCreight’s varies, allegedly with alphabet size as this is what causes the McCreight graphs to rise initially. Determining an approximate relation between input length and alphabet size will not be trustworthy if based on this graph due to only two crossing points being part of the graph, and these two points are both difficult to determine due to deviations in the graphs around these crossings. If one such could be determined, however, one should be mindful of the fact that it would be strictly tied to our implementations regardless, and have little to nothing to say about the performances of the algorithms.

On this Figure 44, it is also evident that for alphabet sizes $2^{14}$ and $2^{15}$, McCreight’s stays below Farach’s within the input lengths part of this plot. Recall these two plot lines in the original Figure 43. In this original figure, they did not cross Farach’s at any point either. And the slopes of all of Farach’s when compared to these two McCreight plot lines indicates that it will not happen for any larger inputs from that point onwards either. This essentially means that for our particular implementations, and for any input string of any length drawn from an alphabet of size less than $2^{15}$, we would expect McCreight’s to outperform Farach’s.

### 8.5.2 Relative alphabet sizes

When McCreight’s and Farach’s were compared on input strings of increasing length drawn from an alphabet of fixed size in previous section, we observed that Farach’s was faster for a range of input lengths varying with alphabet size. This range was determined by the initial rise of McCreight’s plot lines, which was claimed to exist
because of an increasing number of unique symbols occurring in the drawn strings, which is a consequence of the shape of the experiment.

It is interesting whether a certain ratio between input length and the number of unique symbols occurring in the input can be estimated at which Farach’s will perform faster than McCreight’s. However, as the number of unique symbols is increasing with input length in an uncontrollable manner in Figure 43, it is unclear whether such a ratio exists. This gave rise to a new experiment, where the ratio between input length and unique symbols contained was forced to stay fixed on a given value.

**Experiment details**

For any given input length, a periodic string, as defined in Section 8.4, was constructed, with the period being a given percentage of the input length, rounded down. Three values of percentages was chosen, namely 10%, 15% and 20%. These were chosen to produce an interesting graph within a practical time frame. Before being given to the suffix tree construction algorithms, the periodic strings were shuffled at random.

**Remarks on this way of constructing data**

By creating input strings in the way just described, the ratio between input length and unique symbols within the input will stay fixed as the input length is increased. The random shuffling is done as to break the influence the structure of the periodic strings will have on the suffix tree structure and therefore also the construction time.

This hopefully makes the experiment comparable to that of Figure 43. However, note that by randomly shuffling a periodic string rather than drawing the string uniformly at random from an alphabet of size a percentage of the input length, the number of occurrence of each unique symbol in the string will only differ by at most one, whereas no such guarantees can be made for the random drawing of strings, only expectations.

This may influence the results and thus the validity of comparisons of these results with those reflected in Figure 43, but this influence is likely to be neglectable and comparison of the two figures still being fair to the extent of drawing the conclusions drawn, as discussed later.

**Initial observations**

At a first glance at Figure 47, it is observed that the higher the percentage is of unique symbols occurring in the input string, the shorter the input length seems to be for the point in which the plot lines for McCreight’s and Farach’s crosses each other. This is in line with Figure 43 where we saw McCreight’s plot line crossing the correspond-
8.5 VARYING ALPHABET SIZES

Figure 47: Running time of McCreight’s and Farach’s with increasing input length. Each algorithm was run 3 times with different percentage of unique characters. We created strings containing exactly 10, 15 and 20 percent unique characters.

Farach plot line for larger input length the larger the alphabet was from which the strings were drawn.

Reflections on plausible 0% and 100% plot lines

The above is likely not true when considering the limits of this idea. Consider a string in which 100% of the symbols are unique occurrences. This is the best-case input for the naive algorithm, on which McCreight’s outperformed Farach’s as evident from Figure 40. Consider the other extreme case, where the unique symbols make up 0% of the input string as the input length grows, where by that we mean a fixed size alphabet such that the ratio between the alphabet size and the input length will eventually converge to 0 for long enough inputs, regardless of choice of alphabet size. This is the case depicted in Figure 43, in which a tendency of McCreight’s outperforming Farach’s for large enough input lengths and onwards is shown for all choices of fixed alphabet size, and with no reason to believe otherwise for all other potential choices of alphabet size. This leads to believe there is some range of uniqueness-percentages for which Farach’s is eventually faster than McCreight’s, and vice versa for the remaining percentages. Unfortunately, this cannot be verified from the experiment at hand, and is left up as future work.

Justification of comparison between Figures 47 and 43

All red plot points in Figure 43 represents the expected construction times for McCreight’s when given an input of some length, drawn from an alphabet of a given size. Each such red plot point positioned
north of some green plot point for the same input length, representing the same expected construction time for Farach’s, is an indication of a ratio between input length and alphabet size for which Farach’s outperforms McCreight’s. This is true to the extend one can argue, that the alphabet size does in fact correspond to the number of unique symbols occurring in a string drawn of this length from an alphabet of this size. Whether this is a fair assumption is indicated by the degree of convergence of the green bars on Figure 45.

As an example of this, consider the plot lines for $|\Sigma| = 8 \cdot 10^4$ for both Farach’s and McCreight’s in Figure 43. They cross each other the second time at input lengths of just below 800,000, from which point onwards, McCreight’s will be faster than Farach’s. It is also worth noting that the number of unique symbols occurring will likely stay largely unchanged as the input is further incremented. It is also fair to assume, that at this point, the drawn strings contain all $8 \cdot 10^4$ characters of the alphabet. This assumption is based on Figure 45 where on the plot for $|\Sigma| = 8 \cdot 10^4$, the green bar corresponding to the expected number of unique symbols occuring in a string of length $81 \cdot 10^4$ is the entire alphabet as the level of the green bars shows clear indications of having converged to $80 \cdot 10^3 = 8 \cdot 10^4$.

What is interesting is that, when looking at Figure 47 for the graphs of the two algorithms on inputs consisting of 10% unique symbols, they cross each other just short of input lengths of 800,000 as well. This crossing point is at a construction time of just above 100 seconds, roughly matching that of the corresponding crossing point on Figure 43, which is found at just above 100 seconds as well. This is because, for inputs of length 800,000, 10% unique symbols translate to $8 \cdot 10^4$.

This same argument applies to input length roughly 430,000 as well, where the graphs for 15% crosses in Figure 47 as well as graphs for alphabet size $|\Sigma| = 2^{16}$ crossing in Figure 43, with $2^{16} \approx 64,5 \cdot 10^3$, which is 15% of 430,000.

Sources of error

Numerous sources of error can be at fault for inaccuracies in the justification for and the comparisons themselves between Figures 47 and 43. These include, but is not necessarily limited to variance in the graphs, the alphabets not being the exact same despite expectations, the differences in symbol frequencies in the two input types and cross-points being loose estimations. We do, however, believe that the numbers add up sufficiently to establish a certain credibility towards the justification of comparison, in spite of these sources of error.
Round off

In Figure 43, the difference in dependence on the size of the alphabet between McCreight’s and Farach’s algorithms were shown to be present in the implementations at hand to a significant degree.

Figure 47 indicates, that for data types being drawn not from an alphabet of fixed size, as was the case in Figure 43, but from alphabets of size proportional to the length of the input, there will be inputs long enough for which Farach’s will perform better than McCreight’s. This, however was questioned whether true for all possible proportionalities of alphabet sizes and input lengths through reflections on prior experiments, namely those for which the results were depicted in Figures 40 and 43.

It also remains unknown for which ratios between alphabet sizes and input lengths Farach’s outperforms McCreight’s. If one was to be found regardless, one should be mindful of this ratio only being a reflection of the relative performances of the implementations of the algorithms. The tendencies shown by the graphs do indicate to us that a such ratio could be estimated through further experimentation.

It would be interesting to experiment with percentages spanning the full range of 0% to 100%, as McCreight’s seemed to outperform Farach’s in both extreme cases, with Farach’s performing better for the three choices of 10%, 15% and 20%. This does not warrant a clear conclusion, yet it does invite for further experimentation.
9.1 DNA

Two sequences were chosen as to represent this type of data, namely the BRCA1 gene and the human chromosome 2, both acquired from GenBank at The National Center for Biotechnology Information [8]. Due to time spent by our implementation producing suffix trees, only excerpts of the chromosome dataset was chosen to construct sufficiently many suffix trees in a reasonable time frame to allow for a satisfying plot.

Experiment details

When dealing with the chromosome 2 dataset in particular, excerpts were selected due to the size of the dataset relative to the performance of our implementations not enabling a construction in a timely manner of the full dataset. Thus, these excerpts were chosen as being prefixes of increasing size of the suffix of the dataset, starting at the 11.000th character. This number was chosen as to be the first recognizable number above the initial 10.142 characters, which are all identical occurrences of “N” indicative of an unknown basepair. We know from the experiment in Section 8.2 that our implementation cannot handle this amount of subsequent identical characters, due to the use of recursion in the algorithm.

Also, the motivation behind the experiment is to disclose of inherent structures in real data and its effect on construction times. These occurrences of “N”, by being a representant of an unknown basepair, naturally cannot be the base of an exploration of effects of the structure of known basepairs.

Besides, we feel this exclusion of the initial 11.000 characters is fair, as there is no value apparent to us in creating a suffix tree over a string of all identical characters besides that of testing performance of an implementation, which has been previously covered.

Observations

When looking at the graphs in Figure 48, McCreight is faster than Farach for all input lengths. This is consistent with the experiments we performed earlier in Section 8.5 for varying alphabet sizes. The tendency we saw was that for alphabet size $|\Sigma| \leq 2^{15}$ McCreight would be faster than Farach for all input sizes. In the DNA sequences the alphabet size was $|\Sigma| = 4$ which is significantly smaller.
Figure 48: Comparing the three algorithm implementations on DNA sequences. The plots look much as to be expected, and is following the same trend as for uniformly randomly drawn data. Testing on different DNA sequences does not seem to affect the running time.
At an initial glance at Figure 48, the graphs seem to both resemble each other as well as the graph depicted earlier in Figure 38. Both relative algorithm tendencies and actual construction times seems to match on these three figures, indicating that the inherent structures in the DNA datasets are of little to no importance to the suffix tree construction algorithms at hand. Farach’s, for instance, constructs a suffix tree on uniformly randomly drawn data with $|\Sigma| = 10$ in roughly $\sim 20$ seconds for input lengths of $100,000$, whereas excerpts of the same length from the human chromosome 2 is constructed in just above this, in approximately $\sim 22$ seconds.

That McCreight’s and the naive algorithm are both faster than McCreight’s is no surprise, as it was established that McCreight’s would likely be faster for all input lengths for alphabets of sizes smaller than $|\Sigma| = 2^{15}$ using these implementations.

The differences in time between the DNA datasets and the uniformly randomly drawn strings are likely due to the increased likelihood of longer LCPs of suffix pairs, whether due to inherent structures or the smaller alphabet size. A comparison with uniformly randomly drawn strings with $|\Sigma| = 4$ along with representative sampling of LCP lengths of suffixes of the DNA datasets compared to the expected lengths of the LCP lengths of randomly drawn data could aid in verifying this.

9.2 ENGLISH TEXT

When exploring English written text, the data consists of more unique symbols than in the previous experiment on DNA data. However, we do not expect this to have much effect on the observed performances in light of the conclusions drawn in Section 8.5 in which it was argued, that McCreight’s would outperform Farach’s for inputs of any length on alphabets of sizes smaller than $|\Sigma| = 2^{15}$.

This expectation further builds on conceptions of the LCPs of suffixes of English written text on average being in orders of magnitudes comparable to those of both DNA data sets and uniformly randomly drawn data with $|\Sigma| = 10$. While the English language obviously have inherent structure, enabling for it to be used as a means of communication, these will not necessarily reflect in the average length of the LCPs of suffixes. For these to be long, the author must use repetitions on both word and sentence-level, as well as be sticking to a simple vocabulary, which we doubt is the case in general written English to a noticeable extent.

Sampling of the average LCPs of suffixes of English written text as compared to that of DNA datasets and the known expected values hereof on uniformly randomly drawn data [14, 15, 24] could support this claim.
Figure 49: Comparing the three algorithms on English text. As expected we observed the same trends as for random data, and DNA sequences.

Experiment details

The text version of novel *Moby Dick* by Herman Melville as found at Project Gutenberg [19] was chosen as a representant of english written text. This version contains text not part of the original novel, and was thus stripped to only include the lines $817 - 23239$ from *Chapter 1* to and including *Epilogue*. While the stripped text is in fact english written text, it was excluded in the hopes of making the experiment relatable to other experiments performed on the novel *Moby Dick* and not necessarily on this particular version.

Observations

When looking at the graphs in Figure 49, the graph once again seems to resemble that of both DNA datasets and uniformly randomly drawn data with $|\Sigma| = 10$. McCreight’s and the naive algorithm follows each other closely, both being significantly faster than Farach’s. As in the experiments with DNA datasets, this is not surprising. When considering the alphabet sizes in these experiments in relation to those in Section 8.5, particularly in the discussion of the zoomed-in graph in Figure 44, it was observed that McCreight’s seems to perform faster for any input lengths when the alphabet size is smaller than $2^{15}$ for our implementations. The alphabet size for our representant for english text, namely the chosen text version of *Moby Dick* as available at Project Gutenberg [19], is $|\Sigma| = 82$, well below $|\Sigma| = 2^{15}$. This alphabet size was determined by counting the number of uniquely occurring symbols in the dataset. Despite this alphabet size being larger, it is still insignificant in relation to the observed
alphabet sizes for which Farach’s and McCreight’s starts to have comparable construction times.

Real data round off

The largely similar behaviours on arguably much different cases of real datasets indicates an equally largely predictable behaviour on all other datasets reminiscent of those featured. We do not feel there is much need to continue amplifying the existence of this predictability of the relative performances of the algorithms until cases of real data significantly different from those included here are envisioned or identified. Such datasets and the existence of these were discussed in Section 10. Due to us not being able to identify these datasets for which different performance patterns could be envisioned, we have chosen to stop the extension of this chapter with more real data types.

9.3 Summary

The following figures shows plot lines from selected experiments for each algorithm, all plotted in the same coordinate system. This is to give an overall impression of how each algorithm, and our implementations hereof, handles diversity in input data with respect to the different attributes of data having been focused in the prior experiments. The degree of which this is in fact indicative of the way in which each of the algorithms handle diverse data types is of course dependent on whether one approves of the attributes of data being the focus of the experiment is in fact in line with the understanding of diversity in data types. One should also be wary of claims based on comparisons of these, as they essentially attempt to encapture general behaviour, and attempts to do so based on different experiments, with not every experiment being a part of every plot. This does not mean that comparisons are invalid, merely that the resulting claims should be viewed in light of the nature of these comparisons.

Observations

In particular, it is observed that the construction time span on the y-axis on Figures 50 and 51 is identical, whereas it is significantly higher on Figure 52. This is due to the sheer difference in overhead of the three implementations, and is in line with the overall trend observed throughout the various individual experiments.

Considering each of the plots individually, Figures 50 and 51 both have plot lines standing out from the general tendency observed, compared with the fact that 52 does not. This tells us that our implementations of McCreight’s and the naive both has certain inputs for which they perform remarkably different from others, whereas Farach’s will
Figure 50: A summary of selected experiments with the naive algorithm implementation. As expected, the majority of the plot lines are grouped up, indicative of the sum of LCPs of suffixes in data in general being relatively short. In both extreme cases of long and short LCPs of suffixes, it is reflected in the performance.

vary more in construction time, as seen by the larger degree of the plot lines fanning out, but be largely unaffected by attributes of the data at hand.

The fact that Farach’s plot lines seem to generally fan out to a higher degree than those of McCreight’s and the naive is likely due to the overhead in the implementation being larger, and thus changes in the amount of work being done as dependent on the particular input will influence the final construction time more. This degree of fan out of the implementation could likely be brought down through iterations of optimization, bringing down the overhead currently in the implementation of the algorithm.
Figure 51: A summary of selected experiments with McCreight’s algorithm implementation. Much as expected, it closely resembles the observed trends in Figure 50, but with the slowest constructions being faster and the fastest slower, due to the optimizations and the overhead, respectively, in the implementation of McCreight’s algorithm.

Figure 52: A summary of selected experiments with Farach’s algorithm implementation. The larger overhead of the implementation amounts to larger construction time differences per increase in work required to build each tree. This reflects in the larger differences in slopes of the plot lines, along with the larger overall construction times, as indicated by the y-axis values.
Uses of Farach’s algorithm

Comparing our implementations of the two linear time complexity algorithms, namely McCreight’s and Farach’s, our experimentations showed Farach’s to outperform McCreight’s for data types in which the occurrence of unique symbols was large enough relative to the input length, where large is seen in relation to the alphabet sizes of the real data also experimented with. This gives rise to considerations of potential uses of Farach’s on real data. We cannot think of a type of data stemming from the real world in which a percentage of the input will be unique symbols, and for which one would want to construct a suffix tree.

Data types for which one can imagine the occurrences of unique symbols being proportional to the input length could be strings of symbols for which each was a representant of a vector of values, a combinatorial configuration or so, for which there is a potential need for a lot of unique representants. However, this is not enough to argue for a use of Farach’s, as one would also have to justify the need for a suffix tree over these data. We cannot find any such example which justifies this, not to say none exist.

In Figure 52, we observed an interesting trend. There seems to be no data type for which Farach’s performs significantly different from the others, of those included in the summary plot. In both Figures 50 and 51 we saw plot lines standing out, both being significantly faster and slower than the trend seen for the majority of the plot lines, which indicates certain features of the data causing significant shifts in performance in either direction.

That no plot lines are standing out in Figure 52 may simply be because the data causing a such behaviour was not part of the experiments performed. Assuming this is not the case, however, one could argue that not having these performance fluctuations as dependent on features of the data given is a sign of Farach’s algorithm performing reliably, regardless of the data type given. This independence of data has its potential uses, one possibly being in the area of benchmarking. We imagine reliable behaviour, i.e. performance not varying significantly based on particular attributes of the data it is given in a test, is an important feature of an algorithm featured in benchmark tests, as the performance will then likely relate more to other benchmark tests performed, potentially on different data types. The implementation presented by us must be revised and optimized
in this case though, as to limit the overhead causing the large fan out of plot lines.

**Implementation optimizations**

The implementations are not perfect, but merely a first take at implementing the three presented algorithms. Future iterations of optimizing could speed up execution times. We imagine the potential speed-ups achieved to likely be larger with the added complexity in McCreight’s and Farach’s relative to the naive algorithm. Thus, the performance gains might be disproportionate relative to one another, but the overall trends in performance observed during these experiments will likely remain, as optimizations will likely only bring down the performances by constant factors.

The values on the axes in the graphs are thus only a reflection of relative performance for these particular implementations

**Representation of children nodes**

A lot of the work performed by our implementation of McCreight’s in particular is in relation to determining whether or not some downward edge of a given node towards a child node starts with some character matching that of the current position of the current suffix being inserted. This is the cause of what is perceived as an alphabet size dependence in Figure 43 as an example.

In our implementation, this is determined by lookup in a *Dictio-


data structure in Python 3.6.1, used by us to map alphabet characters to children nodes towards which the edges that lead down to it starts with this particular character.

As such, the increase in work in the implementation of McCreight’s algorithm is likely to be due to an increase in both the number of dictionaries needed to be constructed, and the size of each. As seen in Figure 45, both the number of internal nodes and the unique symbols occurring in a string grows with an increase in alphabet size. As such, and considering that lookup-operations in dictionaries are based on hashing [21], we believe the construction of an increasing number of objects of potentially increasing sizes to be the cause of the perceived alphabet dependence.

One could imagine a different representation of nodes and/or edges being used, as well as a different strategy for determining whether an edge with a matching character leaves some node. Regardless of these choices, however, we believe the increase in overhead in McCreight’s algorithm w.r.t. keeping track of symbols on outgoing edges will always grow with an increase in number of nodes and the number of (potential) symbols on edges leaving said node. The tendencies

---

1 Described further at [20]
observed in Figure 43, if this claim holds, will therefore show regardless of these choices.

**Maximum recursion depth exceeded**

As described earlier in Sections 8.2 and 8.4, a *maximum recursion depth exceeded* error occurs when constructing trees on data with either very long or very short periods. Recall that the errors occurred when overmerging two trees, and when following suffix links in trees for strings containing periods with a length of more than 25,000 in our case.

Elimination of recursion in favor of an iterative solution is possible if nothing else by keeping a representation of the call stack in a queue in memory instead of having the actual nested recursive calls. Unfortunately, we did not have time to alter our implementation to use this strategy instead, however it is possible to do so, and we imagine we would do as described in the following, had we had the time.

This could be implemented as such: In both implementations we are calling recursively on nodes, either from suffix links or children nodes. Instead of calling recursively, a first-in first-out queue (FIFO) could be utilized, where a node is inserted into this FIFO-queue whenever a recursive call is currently being made. Whenever a node is then being processed, the next node will then be decided by this queue. This way, the order of nodes as decided by the current recursion is maintained, but with nested function calls forming the structure being replaced by a queue.

The annotation of even and odd descendants for which an overmerged node is the LCA is performed on a node whenever a recursive call on that node returns, as of the current implementation. Altering this to instead utilize a queue, with the nodes currently annotated being appended on this queue in an order matching that of the recursive calls now, will make the same annotated nodes available for annotation on each overmerged node, with a result matching that of the current recursive procedure.

By altering the implementation in the above two ways, one will end up with an implementation of Farach’s with recursive calls being substituted entirely by queue data structures. This will eliminate observed occurrences of *maximum recursion depth exceeded* errors entirely.
The aim of this thesis was to engage in and shed light upon various ways of constructing suffix trees, with particular focus on Farach’s algorithm [6] and its largely unexplored performance in practice.

In order to accomplish this, two algorithms besides Farach’s were chosen as to form a foundation for comparisons, from which an impression of their performances relative to one another could be derived. These two were McCreight’s algorithm [16] along with a naive approach to suffix tree construction.

In implementing Farach’s algorithm in particular, we deviated from the descriptions given in [25, 26] in order to achieve the intended behaviour. These deviations were outlined in Section 6.7.

With the implementations of the three algorithms ready at hand, we proceeded to explore their relative performances as measured in time spent constructing a suffix tree when given a variety of datasets. This not only gave us an impression of the relative performances of our concrete implementations of the algorithms, it also opened up for a discussion of the influence of characteristics in data on each of the algorithms and its reflection on time spent constructing the trees in practice.

A particular mention among these discussions is the one resulting from the experiments in which the implementations were given datasets of uniformly randomly drawn strings, drawn from alphabets of various fixed sizes. Not only did it establish the existence of both the dependence and the independence of alphabet size of McCreight’s and Farach’s algorithms, respectively, it also indicated to what degree alphabet dependence is in fact influential on the construction times of McCreight’s as compared to Farach’s in practice, using our particular implementations.

In line with expectations, as based on technical involvement of implementations, and as allegedly claimed in [25], we found our implementation of Farach’s algorithm [6] to generally perform slower than both our implementations of McCreight’s algorithm [16] and the naive approach to suffix tree construction, however with Farach’s outperforming McCreight’s under certain specific circumstances related to alphabet size.

The experiments undergone were largely focused on various characteristics of data and how each algorithm handled these in relation to one another, with real datasets for which the construction of a suffix tree is realistic being drawn in to give an impression of the degree
of presence of these characteristics in real datasets not artificially constructed.

By considering the experiments collectively as a whole, we have supplied a frame of reference of the performance tendencies of the three described algorithms, which will hopefully generalize to data types not featured in this thesis.

Based on this frame of reference of performance tendencies, and the technical involvement required to implement Farach’s algorithm as opposed to McCreight’s or the naive approach, and through a brief discussion of datasets for which Farach’s algorithm would seemingly outperform McCreight’s algorithm, we feel there is little value to be found in an implementation of Farach’s algorithm in construction of suffix trees in practice.

11.1 FUTURE WORK

In order to gain a contemporary reflection of performance of the algorithms, and in particular the widely unknown practical performance of Farach’s algorithm, a comparison with alternative state of the art construction methods is interesting. Common to these implementations, however, seems to be that they take into consideration the disk I/O complexity of the algorithms, due to the large sizes of data on which suffix trees are typically constructed. Farach’s original algorithm, as described in [6], forms the basis of a later publication [5] in which disk I/O is explored, and this results in an alternate take on the algorithm. This I/O optimal and alternate take on the algorithm is likely more interesting to base said comparisons with state of the art implementations on.

This does, however, require a reconsideration of the provided implementation on the differing parts between the original and the alternate take, along with altering the use of recursion to an iterative approach as described in Section 10, and would be an interesting candidate for future work.

Possible candidates for state of the art algorithms include, but is not limited to, the following: TDD[27] TRELLIS[18], DiGeST[2] and B²ST[3]

Our experiment in Section 8.5.2 on relative alphabet sizes hinted at interesting coherences between input length and uniquely occurring symbols for which Farach’s would outperform McCreight’s, but we were not able to increase the number of plotlines in a timely manner to a point where we felt these alleged coherences could be quantified in a meaningful way.

A better understanding of exactly when Farach’s performs better than McCreight’s, particularly for what relations of alphabet size and input length this is the case, is thus interesting. The shape of the experiment resulting in Figure 47, if repeated with more percentages,
could potentially quantify the extent of this dependence. This quantification, however, will be strictly tied to our unoptimized implementations, but will be interesting nonetheless.
BIBLIOGRAPHY


