FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

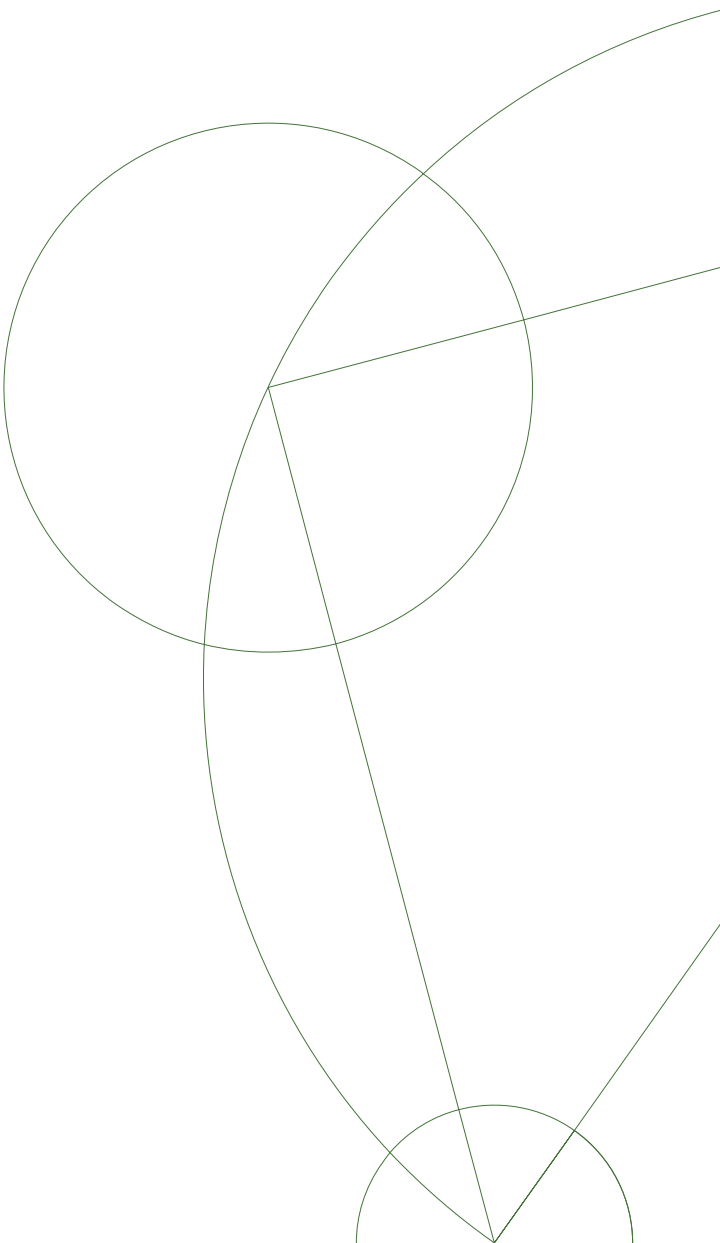
DEPARTMENT OF COMPUTER SCIENCE

Bachelor

Suffix Arrays In Intrusion Detection

April 2017

Mark Roland Larsen <fvg932@alumni.ku.dk>



Supervisors
Michae Thomsenl <m.kirkedal@di.ku.dk>
Troels Larsen <gvl981@di.ku.dk>

Contents

1 Abstract

2 Description

3 Preface

4 Limitations

I følgende opgave arbejdes der på binære træer med typen

5 Introduction

The string matching problem is found in various fields of study [1]. In biology, string matching algorithms significantly aid biologists in retrieving and comparing DNA strings, reconstructing DNA strings from overlapping string fragments and looking for new or presented patterns occurring in a DNA[2]. Text-editing applications also adopt string matching algorithms, whenever the application has to acquire an unambiguous occurrences of a user-given pattern, such as a word in some document[3, 2]. String matching is used in music equipment, AI (artificial intelligence) and in addition, various software applications like virus scanners (anti-virus) or intrusion detection systems, frequently adopt string matching algorithms as a practical tool, to secure data security over the internet [4]. Fundamentally, string matching is a method to find some pattern $P = \{p_1, p_2, \ldots, p_n\}$ in a given text $T = \{t_1, t_2, \ldots, t_m\}$, over some finite alphabet Σ as illustrated in fig. 1 [4].

6 String Matching

Exact string matching is both an algorithmic problem and data structure problem [1]. The static data structure consist of preprocessing some predefined large text $T = \{t_1, t_2, \ldots, t_m\}$, and query some smaller pattern $P = \{p_1, p_2, ..., p_n\}$ [1]. The objective is to preprocess text T and query pattern P in text T in linear time, $O(m), m \in |T|$ [1] and $O(n), n \in |P|$, respectively [1].

Problem:

> Given a pattern P and a long text T, the problem consist of finding all occurrences of pattern P, if any, in text T [2].

The occurrences of pattern $P = \{ana\}$ in text $T = \{banana\}$ are found at $T[1, 3]$ and $T[3, 5]$, as illustrated in Figure 1. Note that pattern P may overlap.

Since most discussions of the exact string matching paradigm, begins with a naive method, this paper adobt the tradition, both presented by Gusfield et. al and by many others [2]. The naive method forms a basic understandig and insight to the more complex exact string mathing algorithms presented in the paper.
The method align left end of P with left end of T and the scan from left to right, comparing characters of P in T, until either there is a mismatch or P is exhausted, in which

[1] See **??** for a description of algorithmic time analysis

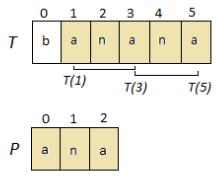


Figure 1: The text T={banana} and pattern P={ana} over the alphabet Σ={abn}. The pattern P occours in T in, at position $T[1]$ and $T[3]$. Notice that occurrences of P may overlap.

case an occurence of P in T is reported. P is then shifted one place to the right, and the character comparison is restarted from the left end of P which repeats until P shifts past right end of T [2].
Let n denote the length of P and let m denote the length of T, then the worst-case time-complexity of the naive method, is $\Theta(nm)$. This is particular clear if P and T consists of the same repeated characters, such that the is an occurence of P in T for each of the first $m - n - 1$ positions.
Since most discussions of the exact string matching problem begin with the naive method. This paper adopt this tradition, as it form a basic insight to the more complex exact string matching algorithms presented later on [2].

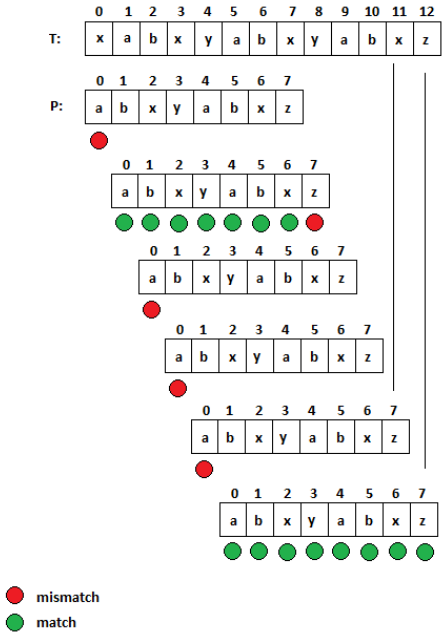


Figure 2: The naive method, where P is shifted one character to the right after each mismatch.

Let pattern P= abxyabxz and let text T= xabxyabxyabxz.

Then the naïve method align left end of P with left end of T and scan from left to right, comparing the characters of P with T, until either two disparate characters are located or P is exhausted, in which case an occurrence of P in T is reported. If a character mismatch happens, P is shifted one place to the right, until P exceeds T, as illustrated in

Figure 2 [2]. The worst-case bound of the naïve method is $\Omega(nm)$, which can be reduced to $\Omega(n + m)$ with the basic idea of shifting P more than one character at a time. This means that the number of character comparisons are reduced, due to P moving through T more rapidly. Some methods even exploit skipping over parts of the pattern after P has shifted, further reducing character comparisons [2].

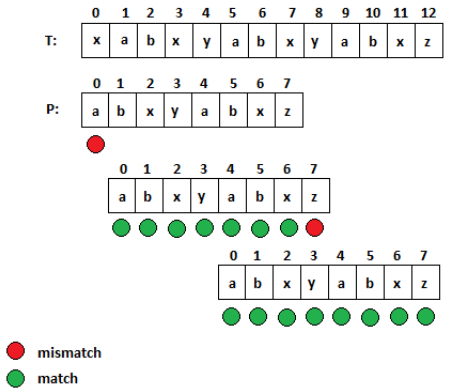


Figure 3: After a mismatch, P is shifted to the next occurrence of a at position 5 in T, moving through T more rapidly

Figure 3 illustrates the idea of shifting P more than one character to the right. At initialization , the left end of P aligns with left end of T, here comparing each character from P with T from left to right.
Let $P[0]$ denote the starting character of P found at position 0, such that $P[0] = a$

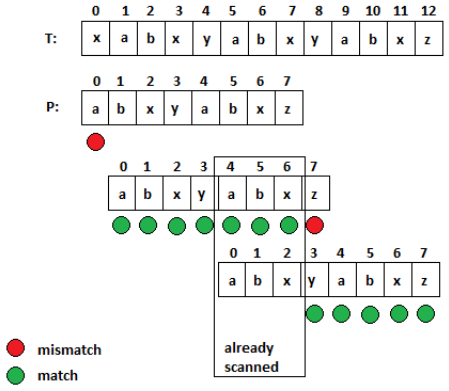


Figure 4: Characters that have already been scanned are stored, so when P is shifted to position 5 in T, abx have already been scanned and can be skipped, and the character scanning is resumed from position 8 and 3, in P and T, respectively.

When comparing characters, if a character in T match $P[0]$, store the location. If a mismatch occur, shift P to the stored location, here position 5 in T and restart the character comparison, as in Figure 3. This is doable for the reason that $P[0] = a$ does not occur in T before position 5, such that $T[5] = P[0] = a$. The method in Figure 3 can be improved further, knowing that the next three characters are abx after P has shifted to position 5 in T. Knowing this, the first three characters are skipped, and character scanning are

resumed from position 8 in T and position 3 in P, as illustrated in Figure 4 [2].
The three methods presented exemplifies the basic idea of comparison based algorithms. More efficient algorithms have been developed, such as the Boyer-Moore and Knuth-Morris-Pratt algorithm, which have been implemented to run in linear time ($O(n + m)time$) [2]. These are without a doubt interesting algorithms to analyze, however this paper merely delivers a short and precise description of the paradigm. Another approach to the comparison based method is the preprocessing approach, where comparisons are skipped by first spending a small amount of time, learning about the internal structure of pattern P or text T. Some methods preprocess pattern P to solve the exact string matching problem, where the opposite approach is to preprocess text T, such as algorithms based on suffix trees [2].

6.1 Suffix trees

The classic application for suffix tree is the substring problem [2, 5], which is both a data structure -and an algorithmic problem [1]. That is, given a long text T over some alphabet Σ, and some pattern P, the substring problem consist of preprocessing T in linear time $O(m)$, and hereafter T should be able to take any unknown pattern P, and in linear time $O(n)$ determine occurrences of P, if any, in T [2]. The preprocessing time is here proportional to the length of text T, and the query is proportional to the length of pattern P [2].
This paper adopts the approach of Gusfield et al., by not applying the denotation of pattern P and text T, in respect to describing suffix trees. By using the general description and denotation of suffix trees, there will be less confusion, since input string can take different roles and vary for application to application [2].

Conceptually a suffix tree is a compressed trie [1].

> **Definition** A trie contains all suffixes of string S, where each edge is labeled with a character from some alphabet Σ. Each path from root to leaf represent a suffix, and every suffix is represented with some path from root to leaf [1, 5].

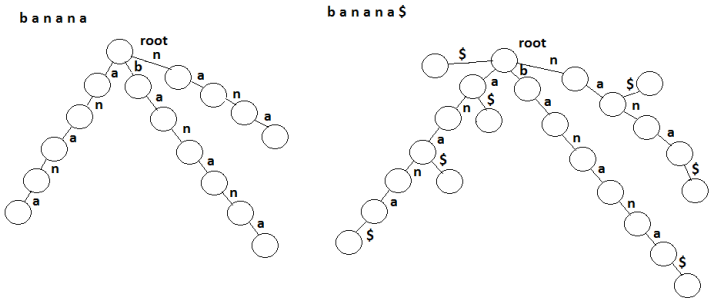


Figure 5: Left is a trie of the string *banana* and the right is a trie of the string *banana*\$.

Figure 5 illustrates two tries, left of the string banana and the right over the string *banana*\$. Note that right trie has the termination character \$ appended to the end. This is due to the fact that the definition of a trie dictates that every suffix is represented with some path from root to leaf. Suffix *ana* in left trie does not have a path from root to leaf,

but appending a termination character to S that exists nowhere else in the string, will eliminate the problem.
Creating a compressed trie, one takes each non-branching nodes and compress them, such that edge-labels from non-branching nodes concatenates into a new edge-label, as illustrated in Figure 6. Here node 1 is a non-branching node, one then concatenate a to n, to form a new edge-label na, deleting the non-branching node [1]. The number of non-branching nodes in a trie is at most the number of leaves. By compressing, we know have that the number of internal nodes is at most the number of leaves, having O(k) nodes total.

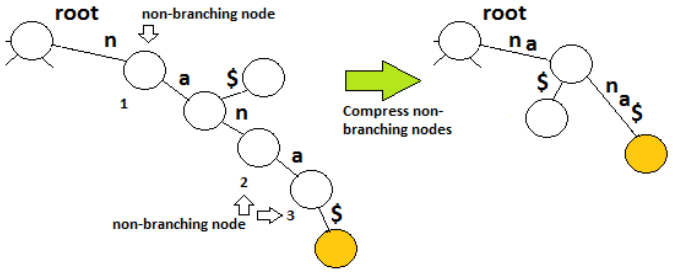


Figure 6: Compressing a trie.

Definition A Suffix tree, T, is a m-character string S concatenated with a termination character \$, that is represented as a directed rooted tree with exactly m leaves, numbered 1 to m. Except the root, each internal node contains at least two children, with each edge labeled with a nonempty substring of S. No two edges exiting a node can have labels beginning with the same character. The concatenation of edge-labels on the path from the root to leaf i, unerringly spells out the suffix of S that starts at position i, such that it spells out $S[i..m]$. The termination character \$ is assumed to appear nowhere else in S, such that no suffix of the consequential string can be a prefix of any other suffix[2].

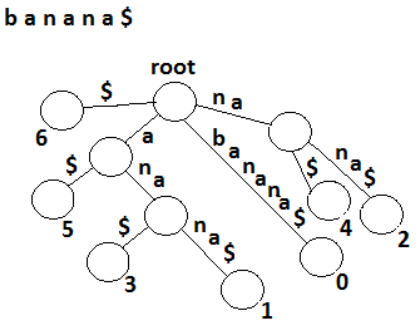


Figure 7: A suffix tree T for string *banana*\$.

The suffix tree for the string *banana*\$, in lexicographical order, is illustrated 7. Each path from the root to a leaf i, unerringly spells out a suffix of S, starting at position i in S. As

an example, leaf numbered 2 spells out *nana*\$, starting at position 2 in the S, such that $S[2..6] = nana\$$. Each node has at least two children, and no two edges exiting a node begins with the same character.
To dive into the substring problem using linear preprocessing time, $O(m)$, and linear search time, $O(n)$ we follow the tradition, and starts with a naive and straightforward algorithm to building suffix trees before verturing into the linear time preprocessing approach [2].

6.2 Suffix Trees To Suffix Arrays In Linear Time

Firsrt, suppose we used a suffix tree as datastructure for an application that resign on a device with abundance of space, but needed it to run a smaller device where the current datastructure would be too large. Then we could convert the suffix tree to the space efficient suffix array, provided that the current operations on the suffix tree is supported by the suffix array [2].

Deffinition

> Let an $edge(v, u)$ in T be lexically less than an $edge(v, w)$ if, and only if the first character in $edge(u, v)$ is lexically less than the first character in $edge(v, w)$ [2]

As no two edges out of v have labels beginning with the same character, edges out of v lexical ordered. So the path from the root of T following the lexically smallet edge out of each node leads to a leaf in T which represents the lexically smallet suffix in T. Then the suffix array SA of T can be constructed in linear time. For suffix tree $T=$ banana in Figure 7, the lexical depth-first traversal visites the nodes 6, 5, 3, 1, 0, 4, 2 in that order [2].
Latter, suppose that we have no datastructure constructed for our new application, but already know that a suffix tree would be too large for our new device. Then it would be easier if we could construct the suffix array without the need of a pre-constructed suffix tree, especially if we could construct it in linear time. Luckily we can, but besides of being more space effecient a more detailed description is needed.

6.3 Suffix Arrays

Suffix array are space efficient alternatives to suffix trees [2, 6]. Before Manber and Meyers in 1990 introduced the first direct suffix array construction algorithm – SACA, suffix arrays were constructed using lexicographical-order traversal of suffix trees [2, 6, 7, 8]. Manber and Meyers made suffix trees obsolete in respect to constructing suffix arrays, and their approach is known as a doubling algorithm, where with each sorting pass, doubles the depth to which each suffix are sorted. This means that suffixes are sorted in logarithmic number of passes, providing a worst case bound of $O(nlogn)$ and $O(n)$ expected, assuming linear sort, reminiscent of Radix Sort [7] and queries can be answered in $O(P+logn)$ with use of Binary Search [2].
With the discovery of four different SACAs requiring only $O-(n)$ time worst case in 2003, the situation drastically changed. SACAs have since been the focus of intense research [7, 8]. In 2005 Joong Chae Na introduced more linear time SACAs, where two stood out, the Ko-Aluru (KA) algorithm for supplying good performance in practice and the Kärkkäinen-Sanders algorithm for its elegance [8].
According to a survey paper, SACAs have to fulfill three important requirements:

1. The algorithm should run in asymptotic minimal worst case time, where linear is an optimal way [8].

2. The algorithm should run fast in practice [8].

3. The algorithm should consume as less extra space in addition to the text and suffix array as possible, where constant amount is optimal [8].

Although no current SACAs fulfill the requirements in an optimal way, research into faster and more space reducing SACAs continued [8]. Later on, in 2009, Nong et al. introduced two new linear time construction algorithms, one which outperformed most known and existing SACAs, called Suffix Array Induced Sorting SA-IS algorithm, guaranteeing asymptotic linear time and almost optimal space requirements [8].

6.4 SAIS - Suffix Array Induced Sorting Algorithm

The SA-IS algorithm is a divide and conquer and recursion algorithm, using variable-length leftmost S-type substrings and induced sorting [6]. In view of the fact that the SA-IS algorithm is unsophisticated to comprehend, implement and guarantees asymptotic linear time construction and close to optimal space, SA-IS has been chosen as the single algorithm for the implementation of a malware detection system and the experiments which follow.

```
SAIS(S, SA)
    (* Step 1 : Initialization & classification*)
    SA <-- suffix array of S
    t <-- type array
    P <-- LMS indicies array
    B <-- bucket array
    Scan S once from either left or right and classify all characters as
        S-type or L-type and place them in t.
    Scan t once from  either left or right and locate all LMS substrings
        in S and put them into P_1
    (* Step 2 : Induced sort LMS-substring)
    Induced sort all LMS substrings using P_1 and B
    Name each LMS substring in S by its bucket index to get a
        new shortened string S_1
    (* Step 3 : Uniqueness - recursive step)
    if T_1 is distinct, hence all characters are unique
       then
            Directly compute SA_1 from S_1
       else
            SAIS(S_1, SA_1)
    (* Step 4 : Induce SA from SA_1)
    Induce SA from SA_1
    return
```

Basic notations

Let S be a string or text of n-characters stored in an array $[0 \ldots n-1]$ and let $\Sigma(s)$ be the alphabet of S.

Let $S\$$ be a string S concatenated with the termination symbol \$, where \$ is not contained in S and is the lexicographical smallest character in S. For S containing concatenation of multiple strings, let $S = S_0\$S_1\$...S_{n-1}\$$, where \$ is the termination symbol for each concatenated string in S, and is the lexicographical smallest character in $S_0, S_1, ..., S_{n-1}$.

Furthermore, S may not be contained in $S_0, S_1, ..., S_{n-1}$. String S is supposed to be concatenated with the unique termination symbol \$, if not explicit stated otherwise [6].

Let $suf(S, i)$ be some suffix in S starting at $S[i]$ running to the termination symbol \$. $suf(S, i)$ is of S-type or L-type if $suf(S, i) < suf(S, i+1)$ or $suf(S, i) > suf(S, i+1)$, respectively [6].

Let $suf(S, n-1)$ be the termination symbol and of S-type [6].

Let $S[i]$ be S-type or L-type, if suf(S,i) is S-type or L-type, respectively [6].

Observation

- $S[i]$ is S-type if $S[i] < S[i+1]$ or $S[i] = S[i+1]$ and $suf(S, i+1)$ is S-type [6].
- $S[i]$ is L-type if $S[i] < S[i+1]$ or $S[i] = S[i+1]$ and $suf(S, i+1)$ is L-type [6].

The properties defined in the observation suggest that scanning from right to left, determing the type of each suffix or character can be done in constant time, $O(1)$, and that the type array t, can be filled in linear time, $O(n)$ [6].

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S:	m	m	i	i	s	s	i	i	s	s	i	i	p	p	i	i	$
t:	L	L	S	S	L	L	S	S	L	L	S	S	L	L	L	L	S

Figure 8: Type array t is filled from right to left

Figure 8 illustrates the filled type array, t, for text S=mmiissiissiippii\$, where text S is scanned from right to left, determining the type of each suffix and character. Going from right to left in Figure 8 we have that $suf(S, 16)$=\$ is a S-type, $suf(S, 15)$=i\$ > $suf(S, 16)$=\$ and is L-type, $suf(S, 14)$=ii\$ > $suf(S, 16)$=i\$ and is L-type and so forth, filling the type array t in linear time.

Let $S[i]$ be a left most S-typeLMS character, if $S[i]$ is S-type and $S[i-1]$ is L-type, and let $suf(S, i)$ be a LMS suffix, if S[i] is a LMS character [6].

Let $S[i..j]$ be a LMS substring if both S[i] and S[j] are LMS characters, and there exists no other LMS characters in the substring, and $i \neq j$ or it is the sentinel itself [6].

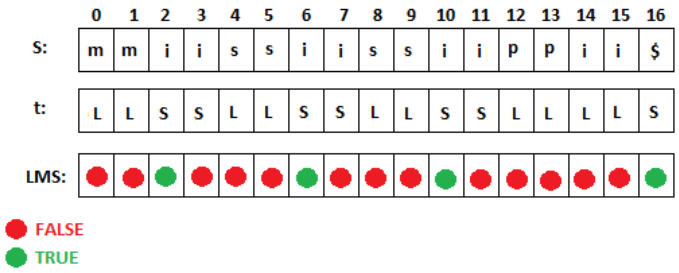


Figure 9: Type array t and LMS array defined for S=mmiissiissiippii$

As Figure 9 exemplify, four LMS characters are defined for S=mmiissiissiippii$, here at position 2, 6, 10 and 16 in S. Furthermore, four substrings and suffixes exists in S, namely S[2..6], S[6..10], S[10..16] and S[16..16], and S[2..16], S[6..16], S[10..16] and S[16..16], respectively. After defining the S-types, L-types and LMS, the induction process of LMS substrings commence.

Deffinition

> Determining the order of any two substrings, the corresponding characters are compared from left to right, comparing their lexicographical values first, and next their types, where S-type is considered highter priority than L-type [6].

Induced sorting LMS substrings

This part address the challenging problem of sorting the variable length LMS substrings. The basic idea is to create a new array, SA, and bucket sort the LMS substring into their equivalent buckets. Each bucket is named corresponding to the alphabet $\Sigma = \{\$, i, m, p, s\}$ in lexicographical order, such that SA contains four buckets, named \$, i, p and s in that order, as shown in Figure 10 [6]. S is scanned from left to right, and indices for each LMS substring is appended to the end of its corresponding bucket in SA. The first LMS substring index is placed at the end of bucket for i, here at position 8 in SA and forwards the bucket end one to the left, hence the bucket end for i now rest at position 7 in SA. This process is repeated until all LMS substring indicies are placed in their buckets [6].

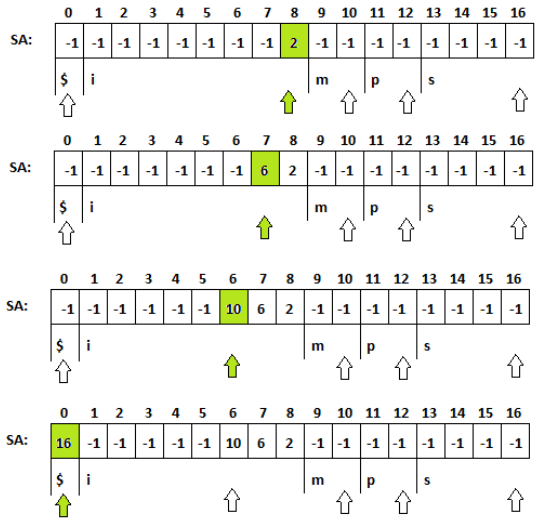


Figure 10: Induced sort of LMS substring

When the LMS substrings are placed, then scan SA from left to right and for each nonnegative value $S[i]$, if $S[i]-1$ is L-type, then place $SA[i]-1$ in the corresponding bucket for $suf(S, SA[i]-1)$, and lastly forward the bucket head one to the right [6].

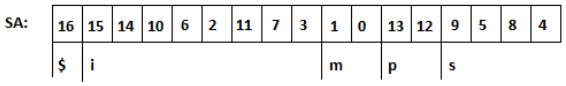


Figure 11: SA after the induced sorting for LMS substring.

Roughly equivalent, when all L-types are placed, scan SA from right to the left for each nonnegative value S[i], if $S[i]-1$ is S-type, then place $SA[i]-1$ in the corresponding bucket for $suf(S, SA[i]-1)$, and forward the bucket end one to the left. The above operations are demonstrated in Appendix B and the final result is displayed in Figure 11 [6].

It is now the matter of determine if all LMS substrings are correctly sorted in SA, hence the uniqueness step in the SAIS algorithm. This is done by scanning SA from left to right, and obtaining each LMS substring, and comparing the lexicographical values and types, and place them in buckets named accordingly to the lexicographical order they appear, starting from 0. So scanning from left to right in SA given in Figure 11 gives the following bucket $B = \{\{0; \$\}, \{1; iippii\$\}, \{2; iissi, iissi\}\}$. The bucket keys are then placed in S_1 in the order as they appear in the original string S, hence $S_1 = \{2, 2, 0, 1\}$ as illustrated in Figure 12. If each character in S_1 is unique, hence does not exists any where else in S, then SA_1 can be computed directly from S_1, else fire the recursive step $SAIS(S_1, SA_1)$. S_1 for S in Figure 12 is not distinct, since 2 exists twice in S_1, hence 2 is not unique, consequently a recursive step, $SA(S_1, SA_1)$, is needed. Before venturing into the recursive step, save the original positions of the LMS substrings as they appear in S_1 into P_1, where $S_1[0] = 2$ points at position 2 in S, $S_1[1] = 2$ points at position 6 in S, $S_1[2] = 1$ points at position 10 in S and finally $S_1[3] = 0$ points at position 16 in S, such

that $P_1 = [2; 6; 10; 16]$ [6].

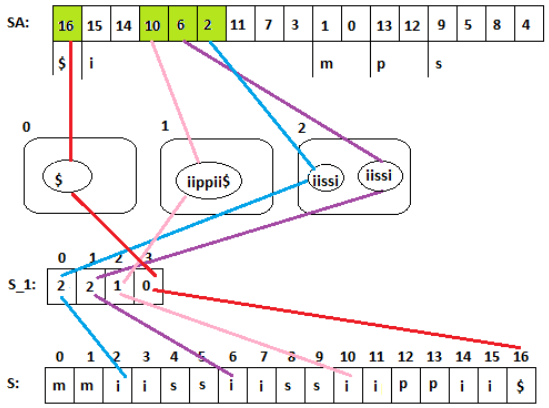


Figure 12: Building S_1 from SA, using the original positions of the LMS substrings in S.

In the recursive step for $SAIS(S_1, SA_1)$, locate S-types, L-types and LMS characters/-substrings and determine if S_1 is distinct. In this case, as demonstrated in Figure 13, there is only one LMS substring in S so the new string S_1 is trivially distinct.

Induce sort SA from SA_1

Either SA_1 has been computed directly from S (if S is distinct) or returned from one or more recursive steps. In either case, SA can be induced sorted from SA_1 using information bound in P_1 [6].

First initialize all indicies in SA with -1 and find the bucket ends. Then scan SA_1 from right to left and place $P_1[SA_1[i]]$ at the corresponding bucket end, and forward the bucket end one item to the left [6].
Then sort L-types by scanning SA from left to right for each non-negative item $SA[i]$. If $SA[i-1]$ is L-type, place $SA[i-1]$ in the corresponding bucket head for SA and forward the bucket head one item to the right [6].
Last, sort all S-types by scanning SA from right to left for each non-negative item SA[i]. If $SA[i-1]$ is S-type, place $SA[i-1]$ in the corresponding bucket end, and forward the bucket end one item to the left [6].

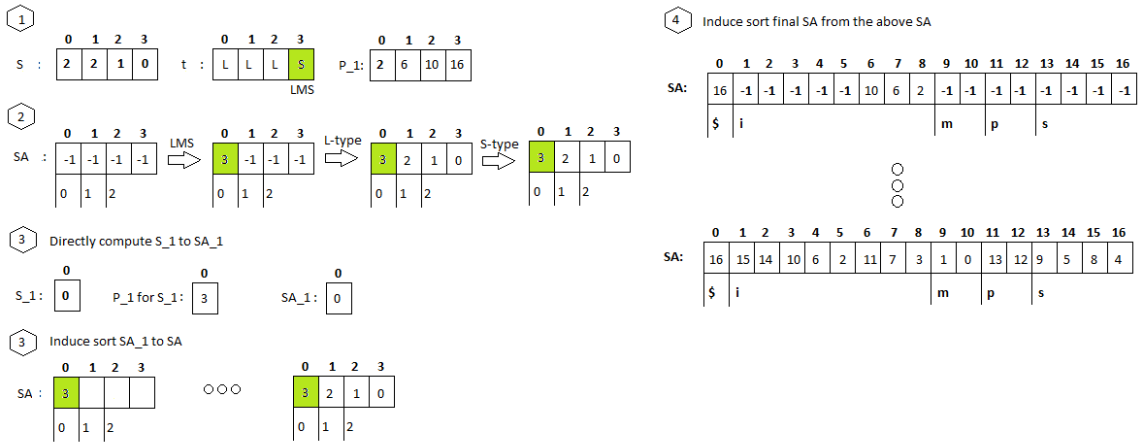


Figure 13: The recursive step $SA(S_1, SA_1)$. First S-types, L-types and LMS characters/substrings for S_1 is localized. Secondly, induced sort LMS substrings, such that $SA_1 = [3; 2; 1; 0]$. Last, scan SA_1 from right to left and place $P_1[SA_1[i]]$ into the buckets end of SA as described for induced sorting LMS Substrings, for each item in SA_1. The final result is displayed in step 4.

This procedure is demonstrated in Figure 34 where SA is induced sorted from S for the text T=mmiissiissiippii\$. SAIS returns the suffix array SA=[16; 15; 14; 10; 6; 2; 11; 7; 3; 1; 0; 13; 12; 9; 5; 8; 4] for text T=mmiissiissiippii\$ which is indeed sorted in lexicographical order as illustrated in Figure 14 [6].
It is now demonstrated how the algorithm works step by step. We now proceed to analyze the SAIS algorithm, here to insure that the algorithm actually returns a suffix array sorted in lexicographical order, for all legal inputs. Furthermore, the core mechanics of LMS-substring sorting is analyzed, giving a precise understanding of induce sorting LMSs, L-types and S-types. We will then prove correctness and completeness.

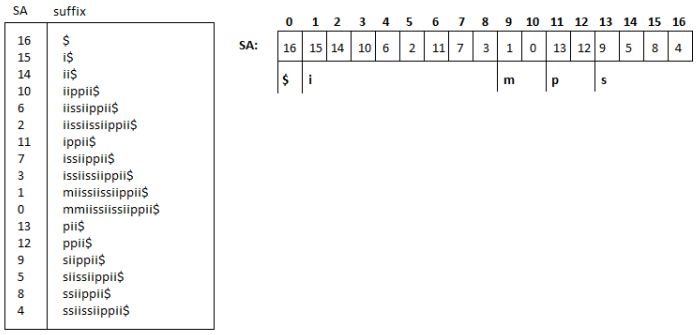


Figure 14: The suffix array for the text T=mmiissiissiippii\$.

OBS! BEVISER SKAL FLETTES IND HER!!!

6.5 Binary Search

6.6 Longest Common Prefix - LCP

The basic task of analysing a single genome, is to characterize and locate repeated elements of the genome. When comparring two or more genomes the task is to find similar subsequences of genomes. This makes repeat analysis to play a key role in the study, analysis and comparison of complte genome. A prefix in an definition for an affix placed before a word. Suppose we have S=abekat, then *abe*, *ab* and *abek* are prefixes, since they are an affixes placed before *kat*, *ekat* and *at*, respectively. The longest common prefix amongst two string *lcp(*S_0, S_1*)* is the length of the longest word both strings share, from left to right.
Let S_0=abekat\$ and S_1=abe, then the longest common word is *abe*, hence lcp(S_0, S_1)=3.

6.7 Burrows-Wheeler Transform

The BWT - Burrow-Wheeler transform, invented by Burrow and Wheeler in 1994, also known as block sorting, is a lossless data compression algorithm and produces a permutation bwt(S) of an input string S, such that S can be reversed from bwt(S), but is easier to compress [9].
BWT is a very powerful tool in data compression and even simple algorithms that implement BWT have good performance and achieve a good compression ratio using relative small space. Furthermore, even more powerful BWT-based compression tools, such as Bzip and Szip are still used today [9].

Besides data compression, BWT have a remarkable and practical property, namely that it can build a data structure which is sort of a compressed suffix array for a input string S [9]. To fully map and understand BWT, and how it succeed to create permutation bwt(S) of an input string S that is easier to compress, this paper gives a precise description of the idea behind cyclic shifts and reversible lossless data compression, before venturing into property of building a data structure resembling compressed suffix array and its importance to the topic at hand.

BWT consist of reversible transformation of input string S denoted bwt(S). This reversible transformation, bwt(S), consist of exactly the same characters as in S over same alphabet Σ, but is usually easier to compress. The idea is to form a conceptual matrix M whose rows consist of cyclic shifts of S sorted in a left to right lexicographical order [9].

Let F denote the first column in M and let F_i denote the i-th character of column F. Almost equivalent, let L denote the last column in M and let L_i denote the i-th character of column L [9].

Then the following properties of M can be defined:

- Every column of M is a permutation of S.
- For $i = 2, \ldots, |2| + 1$, the character L_i is followed by the character F_i in S.
- Any character α, the i-th occurrence of α in F correspond to i-th occurrence of α in L.

We assume that string S is concatenated with the termination symbol \$. We first find each rotation of S and place these in a matrix M, which can be done by repeatedly taking the end character of S and sticking it to the front of S, until all rotations of S is exhausted, as illustrated in Figure 15. Then we sort the rows in lexicopgrapical order from top to buttom as in Figure 16 [10].

F										L
\$	a_0	a_1	a_2	b_0	b_1	a_3	a_4	b_2	a_5	a_6
a_6	\$	a_0	a_1	a_2	b_0	b_1	a_3	a_4	b_2	a_5
a_5	a_6	\$	a_0	a_1	a_2	b_0	b_1	a_3	a_4	b_2
b_2	a_5	a_6	\$	a_0	a_1	a_2	b_0	b_1	a_3	a_4
a_4	b_2	a_5	a_6	\$	a_0	a_1	a_2	b_0	b_1	a_3
a_3	b_4	b_2	a_5	a_6	\$	a_0	a_1	a_2	b_0	b_1
b_1	a_3	b_4	b_2	a_5	a_6	\$	a_0	a_1	a_2	b_0
b_0	b_1	a_3	b_4	b_2	a_5	a_6	\$	a_0	a_1	a_2
a_2	b_0	b_1	a_3	b_4	b_2	a_5	a_6	\$	a_0	a_1
a_1	a_2	b_0	b_1	a_3	b_4	b_2	a_5	a_6	\$	a_0
a_0	a_1	a_2	b_0	b_1	a_3	b_4	b_2	a_5	a_6	\$

Figure 15: Unsorted

Then bwt(aaabbaabaa\$) = aab\$baaaaba, hence the string from L in M_{sorted} read from top to buttom. We notice that the characters tend to stick together in coloumn L. This feature is more obvious with longer strings. As an example, take string S=”tomorrow and tomorrow and tomorrow”, where bwt(”tomorrow and tomorrow and tomorrow”)= ”wwwdd nnoooaatttmmmrrrrrooo \$ooo”, if we were to compress this using run length encoding (RLE) we would get the shortened string ”3w2d2 2n3o2a3t3m5r3o2 \$3o”, hence we would have successfully compressed the original string using bwt(S) and RLE, such that RLE(bwt(S)) shortened the string from 34 to 23 characters, which is a reduction of appoximetly 26 % [10]. It is easy to see how one could reverse RLE compression back to bwt(S), but it is not obvious how one would reverse bwt(S) to the original string S [10].

F										L
\$	a_0	a_1	a_2	b_0	b_1	a_3	a_4	b_2	a_5	a_6
a_6	\$	a_0	a_1	a_2	b_0	b_1	a_3	a_4	b_2	a_5
a_5	a_6	\$	a_0	a_1	a_2	b_0	b_1	a_3	a_4	b_2
a_0	a_1	a_2	b_0	b_1	a_3	b_4	b_2	a_5	a_6	\$
a_3	b_4	b_2	a_5	a_6	\$	a_0	a_1	a_2	b_0	b_1
a_1	a_2	b_0	b_1	a_3	b_4	b_2	a_5	a_6	\$	a_0
a_4	b_2	a_5	a_6	\$	a_0	a_1	a_2	b_0	b_1	a_3
a_2	b_0	b_1	a_3	b_4	b_2	a_5	a_6	\$	a_0	a_1
b_2	a_5	a_6	\$	a_0	a_1	a_2	b_0	b_1	a_3	a_4
b_1	a_3	b_4	b_2	a_5	a_6	\$	a_0	a_1	a_2	b_0
b_0	b_1	a_3	b_4	b_2	a_5	a_6	\$	a_0	a_1	a_2

Figure 16: Sorted

Recall that bwt(aaabbaabaa) = aabbaaaaba and lets introduce LF-mapping, which pro-

vides a subscript (rank) for each character in S, hence each chraracter in S is given a number, equal to the number of times that character accoured previously in S.This proceedure is called S-ranking, since we rank accordingly to S. Ranks are already provided in Figure 16. Notice that the first coloum F and last coloum L have characters that occour in the same order, which is represented by color in Figure 17. This is actually not surpricing feature, since occourences of character c in S are sorted by its right-context in both F and L [10].

F	L
\$	a_6
a_6	a_5
a_5	b_2
a_0	\$
a_3	b_1
a_1	a_0
a_4	a_3
a_2	a_1
b_2	a_4
b_1	b_0
b_0	a_2

Figure 17: Characters in same order

Suppose we want to decompress and find the original string of RLE = 2ab\$b4aba, using *bwt*. First we start with the trivial step expanding the RLE compression to get bwt(aab\$baaaaba). Then we count the appearences of each character in $bwt(aab\$baaaaba)$ and place these in a coloum F in matrix M, in lexicographical order. Put bwt(aab\$baaaaba) in a coloumn L in matrix M and re-rank according to the number of time each character occour in bwt(S) for both F and L, which we will refer to as B-ranking - as in Figure 18.

F	L
\$	a_0
a_0	a_1
a_1	b_0
a_2	\$
a_3	b_1
a_4	a_2
a_5	a_3
a_6	a_4
b_0	a_5
b_1	b_2
b_2	a_6

Figure 18: B-ranking

We can now reverse bwt(aab\$baaaaba) using matrix M. Start at the first row of M, which have \$ in the first coloumn, and since rows are rotations of the original string S, the coloum to the right of \$, F, must contain the character to the left of \$ in S: a_0 in

this case. Hence, we are building the original string from right to left, starting with the termination symbol \$. Now we are in the first row of L containing character a_0, we then find a_0 in F, which mean that the next character in the original string S must be to the right of this index. Continiuing this approach, we end with the string S=aaabbaabaa\$ which is indeed the original string, as demonstrated in Figure 19.

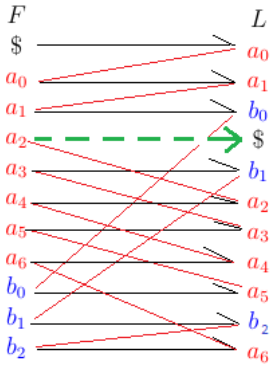


Figure 19: From F–> L to S

The FM-Index provide an opportunity for searching in compressed *bwt* files without full decompressing, which is important for the "Big Data" era of sequencing. We can search in a compressed $bwt(S)$ file using FM-index. Suppose that we want to find an occurrence of pattern P=aab in $bwt(S)$=bc\$ababaaa, so we build a FM-index, as illustrated in Figure . We search P from right to left, and find all occurrences of b in F. Luckily these are grouped together as part of the design and can be found in index 6 through 8 in F. Next we look at index 6 through 8 in L, and find any characters $c = a$, since b is preceded by a in P. Index 7 and 8 in T both have a's, so we look at the ranks of these, and locate the same characters in F with these ranks, which is in index 3 and 4. Then we find occurrences of a at index 3 and 4 in L, since a is preceded by a in P. P is now exhausted, and we can therefore confirm that there is an occurrence of P=aab in $bwt(S)$=bc\$ababaaa. This searching approach find the number of all occurrences of P in $bwt(S)$, but it does not tell us where in the text these occur. FM-index using bwt is a sort of a suffix array, but is easier to compress.

6.8 Application & Operation

We have seen two static datastructures, here the suffix tree and the space efficient suffix array, both which can be constructed in linear time complexity and searched in O(n) and O(n log m), respectively. Now we tend to the question regarding applications and operation, here if suffix trees and suffix arrays can be used in the same applications and supports equal operations.
The suffix tree is undouptly one of the most important datastructures in string processing and comparative genomics and once constructed, can efficiently solve a myriad of string processing problems, as demonstrated by Gusfield with almost 70 pages devoted to applications on suffix tress [2, 11]. The application demonstrated by Gusfield can be classified into three kinds of traversals [11]:

- a buttom-up traversal of the complete suffix tree

- a top-down traversal of a subtree of the suffix tree
- a traversal of the suffix tree using suffix links

Figure 20 shows some of the application discussed in Gusfield, with their traversal kind. Although suffix trees are asymptotically linear and plays a huge role in datstructure algorithmics, they are not as widespread in todays software application as expected. There are a few reasons for that. Space consumption of suffix tree are large and act as a bottleneck in large scale applications, since they require 20 bytes per input character. Moreover, it suffers from poor locality of memory references, which causes significant effeciency loss on cached processor architectures and makes it difficult to store in secondary memory [11]. In several geonem analysis and geonem comparison applications the above problems have been identified for suffix trees. But as seen, a more space effecient datastructure exist, hence the suffix array which only consumes $4n$ bytes per input character[11].
Mohamed Ibrahim Abouelhoda *et al.* [11] show that *every* algorithm that uses suffix trees as data structure, can systematically be replaced with an enchanced version of a suffix array (enchanced suffix arrays) and furthermore, solve the same problems in same time complexity. Enchanced suffix array is a datastructure consisting of the suffix array, and some other information or additional tables. Futhermore, enchanced suffix arrays are fast and easy to implement [11].

Application	Type of tree traversal		
	Bottom-up	Top-down	Suffix-links
Supermaximal repeats	√		
Maximal repeats	√		
Maximal repeated pairs	√		
Longest common substring	√		
All-pairs suffix-prefix matching	√		
Ziv–Lempel decomposition	√		
Common substrings of multiple strings	√	√	
Exact string matching		√	
Exact set matching		√	
Matching statistics		√	√
Construction of DAWGs		√	√

Figure 20: Application on suffix trees and their traversal kind [2, 11].

At this time it is now clear how every algorithm using a sufix tree, can be systematically replaced by an algorithm based on suffix arrays [11]. So we look at how enchanced suffix arrays conquer the three kinds of travesals which were classified earlier and whos applications we see in Figure 20.

Buttom-up traversal

We will first look at problems that are usually solved with buttom-up traversal, and show these can be solved with enchanced suffix arrays, using maximal repeated pairs and Zip-Lempel decomposition of a string, as examples [11].
Maximal repeated pairs plays an important role in genome analysis, and the algorithm presented by Gusfield was implemented in the *REPuter*-program [2, 11]. *REPuter*-program uses maximal repeated pairs to find approximate repeats in $O(\|\Sigma\|n + z)$ time, where z is the number of maximal repeated pairs and is based on space effecient suffix trees [11]. Repetitative structures are seen in the field of biology (but not limited to), where genetic mapping requires the identification of certain markers in a DNA that is variable between individuals, called tandem repeats. What varies, is the number of times a substring repeats in an array, refered to as *variable number of tandem repeats* (VNTR) [2]. The

VNTR markers are used during the genetic level to search for specific defective genes or used in forensic DNA fingerprinting, since a small set of VNTR can uniquely charactirize an individual in a population [2].

Definition
A maximal pair in a string S is identical sbstring pairs α and β such that the character at the immidiate left (right) of α is different from the character to its immidiate left (right) of β. So extending α or β in any direction would destroy the equality of both strings [2].

Definition
A maximal pair (or maximal repeated pairs) is given by the triple (p_1, p_2, n'), where p_1 and p_2 is starting positions for two substrings and n' is their length. We define $R(S)$ as the set of all triples of maximal pairs in S [2].

Given string $S = xabcyiizabcqabcyrxar$, there are three occurrences of the substring *abc*. The first and third occurrences of *abc* do not form a maximal pair, but the first and second form the pair (2, 10, 3), and the second and third forms the pair (10, 14, 3). Note that the definition allow maximal pairs to overlap each other, so to model that case we assume that S has a symbol attacted to the start and end that exists nowhere else in S [2].

Definition
A maximal repeat α is defined as a substring of S that occours in a maximal pair in S, hence α is a maximal repeat in S is there is a triple $(p_1, p_2, |\alpha|)$ $\in R(S)$ and α occurs in S at startposition p_1 and p_2. We define $R'(S)$ as maximal repeats in S [2].

In the given string $S = xabcyiizabcqabcyrxar$, both *abc* and *abcy* are maximal repeats. Gusfield presents a algorithm using suffix trees, that finds all maximal pairs in $O(n)$ time for a string of length n and we presented the definition for such pairs [2].
To compute maximal pairs, the implementation presented by Mohamed Ibrahim Abouelhoda *et al.* [11] requires three tables: *suftab*, *lcptab* and *bwttab*. The *bwttab* contains the Burrows and Wheeler transformation (Section 6.7), *lcptab* contains the longest common prefix (Section 6.6, though here it is a tree representation), *suftab* is the suffix array (Section 6.3 and 6.4)[11]. These tables are accessed in sequencial order, which leads to an improved cache coherence and reducing running time, such that maximal repeated pairs can be computed in $O(|\Sigma|n + z)$ time [11]. Mohamed Ibrahim Abouelhoda *et al.* [11] and Kasai *et al.* [12] proved that it is posible to compute maximal repeated pairs with suffix arrays and some additional information. We will not dwell into the algorithm for computing maximal repeated pairs, we use this to emphecies that suffix arrays can be used in the same applications which usauly solve problems with buttom-up traversal on suffix trees. Although as an example, we will show how to compute the Ziv-Lempel decomposition using suffix arrays, which is a lossless compression algorithm [13].

$S[i]$	a	c	a	a	a	c	a	t	a	t	\$
i	0	1	2	3	4	5	6	7	8	9	10
s_i	0	0	0	2	0	1	0	0	6	7	0
l_i	0	0	1	2	3	2	1	0	2	1	0

aca ⇩ aacatat\$

$S[i]$	a	c	a	a	a	c	a	t	a	t	\$
i	0	1	2	3	4	5	6	7	8	9	10
s_i	0	0	0	0	0	1	0	0	6	7	0
l_i	0	0	1	1	3	2	1	0	2	1	0

Figure 21

We follow the example conducted by Mohamed Ibrahim Abouelhoda *et al.* doing a Ziv-Lempel decomposition of the string S=acaaacatat\$, but with a minor correction. Mohamed Ibrahim Abouelhoda *et al.* claim that there exist a prefix of i=3 of S where l_i=2 is the longest prefix of S[$i \dots n$] that is also a substring of S starting at some position $j < i$, hence S[$0 \dots i-1$]. So S[$3 \dots n$] = aacatat\$ and S[$0 \dots$ 2]= aca, which must mean that the longest prefix of S[$3 \dots n$] which is also a substring of [$0 \dots$ 2] have to be a, with a length of 1, hence l_i=1 and s_i=0 and not l_i=2 and s_i=2 as claimed. This correction should yield the Ziv-Lempel decomposition in Figure 22 [11].

B	1	2	3	4	5	6	7	8
i_B	0	1	2	3	4	7	8	10
B-th block	a	c	a	a	aca	t	at	\$

Figure 22: Ziv–Lempel decomposition of S=acaaacatat [11].

An interesting application of suffix trees is the *lca* (Lowest Common Ancestor) problem, that is, finding the lowest common ancestor of node i and j in tree T. Lowest common ancestor was first obtained by Harel and Tarjan (1984, published online 2006 [15]) and later on simplified by Schieber and Vishkin (1988, published online in 2006 [16])[2].
Lowest common ancestor is an interesting application given that it is used in application as exact matching with wild cards and the k-mismatch problem, amongst others [2]. More interesting is the fact that *lca* of leaves i and j identifies the longest common prefix of suffixes i and j, which will be discussed later on.
By consuming linear time amount of preprocessing a suffix tree, that is a rooted tree, any two nodes can be identified and their *lca* can be found in constant time, $O(1)$ [2, 14]. This paper will not dwell into the different linear time preprocessing algorithms for the *lca* predicament, but delivers an overview and clarification of the problem by introducing a simpler but slower algorithm. (maybe linear in the appendix?).

> **Definition** In a rooted tree T a node u is an ancestor of node v, if u is an unique path from the root to v [2].

> **Definition** In a rooted tree T, the lowest common ancestor of two node u and v, is the deepest node in tree T that is an ancestor of both u and v [2].

Let's suppose for simplification that an application is allowed preprocessing time of an upper bound of $\theta(nlgn)$, which is an acceptable bound for most applications [2]. Then, in

the preprocessing state of tree T, perform a deept-first traversal of tree T and create a list L of nodes in order as they are visited. Then locating the lca of node 2 and 8, $lca[2, 8]$, in fig. 23, one only have to find any occurrences of 2 and 8 in L. Then take the lowest value in interval between $L[1] = 2$ and $L[12] = 8$. This value is the lowest common ancestor for node 2 and 8 in T, $lca[2, 8] = 1$.

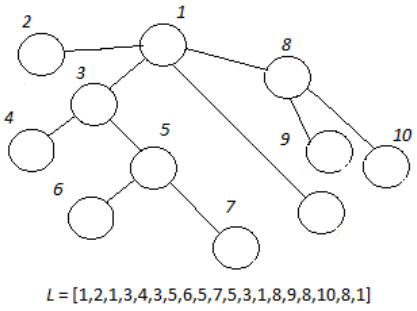


Figure 23: Rooted tree - deept-first travesal with $L = [1, 2, 1, 3, 4, 3, 5, 6, 5, 7, 5, 3, 1, 8, 9, 8, 10, 8, 1]$

Ibrahim Abouelhoda *et al.* did an experiment with two different programs computing maximal repeated pairs, using different files (listed here **??**)[11]. *REPuter* is based on suffix tree and *esarep* is based on enchanced suffix arrays [11]. The result is displayed in Figure 24. The program *esarep* used almost half of the space required for *REPuter* and in this case 2-3 times faster and in 4-5 times faster. This comparison emphzise the advantages of enchanced suffix arrays over suffix trees [11].

ℓ	Running time for *E. coli* ($n = 4{,}639{,}221$) in sec.					Running time for *Yeast* ($n = 12{,}156{,}300$) in sec.				
	maximal repeated pairs			*esasupermax*		*maximal repeated pairs*			*esasupermax*	
	#reps	*REPuter*	*esarep*	#reps		#reps	*REPuter*	*esarep*	#reps	
20	7799	3.28	0.79	899	0.16	175455	9.71	2.23	6432	0.47
23	5206	3.28	0.78	642	0.15	84115	9.63	2.16	4069	0.47
27	3569	3.31	0.79	500	0.15	41400	9.72	2.14	2813	0.45
30	2730	3.30	0.80	456	0.15	32199	9.69	2.14	2374	0.46
40	840	3.29	0.79	281	0.15	20767	9.57	2.13	1674	0.44
50	607	3.29	0.79	196	0.14	16209	9.64	2.12	1354	0.44

ℓ	Running time for Hs21 ($n = 33{,}917{,}895$) in sec.				
	maximal repeated pairs			*esasupermax*	
	#reps	*REPuter*	*esarep*	#reps	
20	40193973	54.63	24.00	188695	1.50
23	19075117	51.78	14.62	138523	1.44
27	8529120	47.97	9.88	98346	1.39
30	4787086	46.54	8.15	77695	1.34
40	732822	45.06	6.21	35719	1.23
50	149482	44.33	5.85	16392	1.19

Space requirement in megabytes

	REPuter	*esarep*	*esasupermax*
E. coli	61	31	31
Yeast	160	83	83
Hs21	446	227	227

Figure 24: Table for computing maximal repeated pairs and supermaximal repeats. Running times are in seconds and space requirements are in megabytes. The number of repeats of length $\geq$ l is displayed in column titled *#reps*. Space requirement is independant of l [11].

Top-down traversals

Exact string matching is usualy computed in a top-down approach when using suffix trees as datastructure as illustrated in Figure 20. The starting positions of P in T are is displayed on every leaf in the subtree below the point of the last match, demonstrated in Figure 25. We match the characters of P down the unique path in T until P is exhausted or a character in P can not be matched. So if P is fully matched from the root along some path in T, we can find all occurrences of pattern P in T by buttom-up travesing the subtree below the end of the matching path and note the leaf indexs encountered. We can do this because every internal node has at least two children, so the number of leaves is proportional to the number of traversed edges [2].

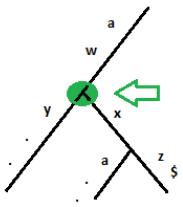


Figure 25: Suffix tree T for S=awyawxawzz, where there are three occurrences of the pattern P=aw in T, with their positions accordingly [2].

Another application exploiting the top-down approach is the exact set matching, where the problem consist of locating all occurrences from a set of patterns P_{set} in some string S, where the set is sent as an input all at once [2].
We define a *keyword tree* as:

Definition
The *keyword tree* T_{key} for the set of patterns P_{set} is a rooted directed tree, which must satisfy three conditions:

- All edges are labeled with exactly one character
- Any two labels comming from the same node, must have distinct labels
- Any pattern P_i in P_{set} maps a node v of T_{key} in a way that all characters from the root to node v spells out pattern P_i [2].

Suppose that we have a set of patterns $P_{set} = \{$potato, poetry, pottery, science, school$\}$ and its keyword tree defined as illustrated in Figure 26. Since no two labels comming from the same node have identical labels, we can use the keyword tree to search for any occurrences of P_{set} in string S. Notice that we preprocess P_{set} into a *keyword tree* P_{key}, such that that we can find all occurrences of patterns in P_{set} in S by taking each position p in S and follow the unique path from r in T_{key} which matches a substring in S starting at character p. The dictionary problem is one of which where the set matching effeciently solves the problem. In this problem, a known set of strings, forming a dictionary is preprocessed to which a sequence of individual string is presented to the dictionary. Each of these strings is then either contained or not contained in the dictionay. The keyword tree solves ecactly that [2]. Mohamed Ibrahim Abouelhoda *et al.* [11] proved that any application that uses top-down traversals on suffix trees, can be solved using suffix arrays with some extra information [11]. Mohamed Ibrahim Abouelhoda *et al.* conducted experiments

for answering enumeration queries, with three different programs: *streematch* (linked list representation of suffix trees), *mamy* (suffix arrays with additional buckets) and *esamatch* (based on enchanced suffix arrays) [11].

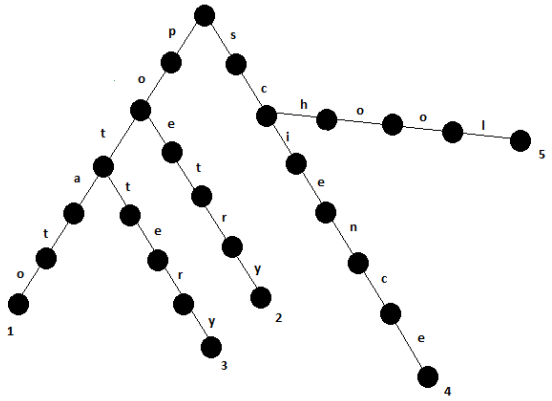


Figure 26: The *keyword tree* T_{key} over the set of patterns P_{set}={potato, poetry, pottery, science, school}.

The expriments were conducted on five different files listed below:

labelFILESE. coli : The complete genome of *Escherichia* bactirium (DNA) - Σ=4, length 4,639,221 Yeast : The complete genome of *Saccharomyces cerevisiae* (DNA) - Σ=4, length 12,156,300 Hs21 : A complete collection of protein sequences - Σ=4, length 33,917,895 Swissprot : Complete collection of protein sequences - Σ=20, length 29,165,964 Shaks : A collection of the complete work of William Shakespear - Σ=92, length 5,582,655 bytes

As illustrated in Figure 27 the program using enchanced suffix array *esamatch*, outperforms all other programs in both time and space consumption, except the file *Sharks*, which is explained by the large alphabet size. The program *esamatch* can thereforee compete with the other programs for small alphabets [11].

File	Running time for *minpl* = 20, *maxpl* = 30			Running time for *minpl* = 30, *maxpl* = 40		
	streemach	*mamy*	*esamatch*	*streemach*	*mamy*	*esamatch*
E. coli	9.47	5.56	**4.48**	9.63	5.70	**4.69**
Yeast	12.42	8.26	**5.37**	12.56	8.46	**5.80**
Hs21	20.15	12.50	**7.23**	20.43	12.69	**7.30**
Swissprot	41.78	9.55	**6.22**	40.80	10.09	**6.25**
Shaks	15.61	**4.29**	72.44	15.78	**4.37**	66.60
	Running time for *minpl* = 40, *maxpl* = 50			Space requirement		
	streemach	*mamy*	*esamatch*	*streemach*	*mamy*	*esamatch*
E. coli	9.86	5.87	**4.85**	56	40	47
Yeast	13.34	8.63	**5.74**	146	106	120
Hs21	21.22	12.88	**7.61**	407	296	327
Swissprot	42.96	9.83	**6.39**	320	288	281
Shaks	15.88	**4.49**	67.16	52	48	60

Figure 27: Space requirements and running times in megabytes and seconds, respectively. The programs did one million enumeration queries searching for exact patterns in the input strings. Minimal (*minpl*) and maximal (maxpl) are the minimal and maximal size of the patterns, respetively [11].)

Suffix links

Size of suffix trees can be reduced with the help of matching statistics, which is needed in more complex problems than exact string matching. Matching statistics are central to a fast approximate matching method designed for rapid database searching, it furthermore provide a bridge between exact matching methods and appromate string problems [2].

- **Notation** Let ms(i) denote the matching statistics for some i in string S

Preprocess suffix tree T for the fixed short string S_p and keep suffix links duing the construction of the tree. We define a suffix link as:

> **Definition** Let an arbitrary string be denoted by $x\alpha$, where x is a single character and α a possible empty substring. For an internal node v with a path label $x\alpha$, if there exists another node $s(v)$ with the path label α, then a pointer from v to $s(v)$ is a suffix link.

The suffix can then be used to find sm(i) for all i in S. Let S_p be a string of length m, then a matching statistics ms$i$ of S_p for all i in S, is a table of pairs (l_p, p), where $0 \leq j \leq m-1$ and the following holds:

- $S_P[i \dots j + t_p - 1]$ is the longest prefex of $S_p[j \dots m-1]$ that is a substring of S.
- $S_p[j \dots j + j + l_p - 1] = S[p_j \dots p_j + l_j - 1]$

Supose we have S= cacaccc and S_p=caacacacca, then we would construct the suffix tree for S keeping the suffix links. Then for each position p in S_p we would find the length of the longest common prefix starting at some position in S. If $S_p[0]$=caacacacca, then the longest common prefix have length of 2 and accour in position 0 in S. Next one is $S_p[1]$=aacacacca, where the length of the longest commmon prefix is 1 in $S[0]$ and $S_p[2]$=cacacca with LCP length of 4 occouring in $S[1]$. Continuing this approach we get the matching statistic table in Figure 28 [11].

j	0	1	2	3	4	5	6	7	8	9
(l_j, p_j)	(2, 0)	(1, 1)	(4, 1)	(6, 0)	(5, 1)	(4, 2)	(3, 3)	(2, 4)	(2, 2)	(1, 3)

Figure 28: Matching statistic table for S= cacaccc and S_p=caacacacca [11].

The algorithm provided by Chang and Lawler [11] for computing matching statistic used suffix links and could solve the problem in O($n + m$) time. Later on Mohamed Ibrahim Abouelhoda *et al.* [11] proved that any problem that used suffix links with a suffix tree datastructure, could be solved using suffix arrays with some extra information in same time complexity as the algorithm presented by Chang and Lawler[11]. Mohamed Ibrahim Abouelhoda *et al.* performed an experiment where two programs computed the matching statistics on a pair of genomes, using a suffix tree and an enchanced suffix array as data structure, respecively [11]. The result displayed in Figure 29 revealed that *esams* consumed 30-40% less space in respect to *streems*, although the latter were up to three times faster [11].

Genome pair	Total length	*streems*		*esams*	
		time	*space*	*time*	*space*
Streptococuss 2	4,199,453	4.1	30	11.1	21
E. coli 2	10,107,957	13.3	65	18.9	43
Yeast 2	24,690,687	41.0	170	43.4	109
Human 2	67,739,601	169.2	472	314.0	294

Figure 29: Running time in seconds and space consumption in megabytes for matching statistics. The program *streems* uses suffix tree as data structure, where the tree construction time is not included. The program *esams* uses enchanced suffix array as datastructure, here a suffix array with some extra information [11].

6.9 SAIS-FLS - Space requirement reduction for fixed length strings

Suppose that string S consists of fixed length strings concatenated together, where each string $S_0, S_1, ..., S_{n-1}$ is terminated with the sentinel \$ such that $S = S_0\$S_1\$...S_{n-1}\$$.

Let $S = S_0\$S_1\$...S_{n-1}\$$ consist of concatenated fixed length distinct strings, such that $|S_0| = |S_1| = \ldots = |S_{n-1}|$ and each string in S is terminated by the sentinel.

Let the length of pattern P, $|P|$, be same length as the fixed length distinct string in S, such that $|P| = |S_0| = |S_1| = \ldots = |S_{n-1}|$.

Suppose that the string $S = jazz\$fuzz\$quiz\$$ is given and suffix array SA for S has been computed, such that $SA = [14, 4, 9, 1, 5, 12, 0, 10, 11, 6, 13, 3, 8, 2, 7]$ as illustrated in Figure 30.

S=jazz$fuzz$quiz$

SA	Suffixes
14	$
4	$fuzz$quiz$
9	$quiz$
1	azz$$fuzz$quiz$
5	fuzz$quiz$
12	iz$
0	jazz$fuzz$quiz
10	quiz$
11	uiz$
6	uzz$
13	z$
3	z$fuzz$quiz$
8	z$quiz$
2	zz$fuzz$quiz$
7	zz$quiz$

Figure 30: Suffix array and suffixes for S=jazz$fuzz$quiz$

By means of the Binary Search algorithm, suppose we want to find pattern $p_0 = jazz\$$, $p_1 = fuzz\$$ and $p_2 = quiz\$$ in the suffix array for S, such that $SA[i]$ pattern$p_0 = jazz\$$,$p_1 = fuzz\$$ must be a suffix of $T[SA[i]]$. Then $p_0 = jazz\$$ is a suffix of $T[SA[0]]$, $p_0 = fuzz\$$ is a suffix of $T[SA[5]]$ and $p_0 = quiz\$$ is a suffix of $T[SA[10]]$. Now suppose, that we are only interested in exact matching, and do not care for unnecessary suffixes, then notice that we can match all fixed strings in S, with merely three indices in SA for S, which leads to 12 indices in SA for S that are never used when exploiting exact string

matching.

> Let $N_D S$ denote the number of fixed length distinct strings in $S = S_0\$S_1\$...S_{n-1}\$$, where n is number of characters in S.

This paper introduce an algorithm that reduce suffix array size for fixed length exact matching, from $O(n)$ to $O(N_D S)$ space complexity with linear time construction.

```
SAIS–FLS(string S, array SA, int len)
        let SA_FLS = new array[int]()
        for i=0 to i < length(SA) - 1 do:
                if(SA[i] NOT EQUAL TO length(S) - len
               && S[SA[i] + len] EQUALS ''$'')
                        then PUT i in SA_FLS
        return SA_FLS
```

Suppose that the length of the strings in S, len, is known, then scan SA once, from left to right, and find any index where $T[SA[i] + len] = \$$ and add the elements to the new array SA-FLS in $O(m)$ time, where m is the length of SA. Constructing the new suffix array for S using SAIS-FLS, all unnecessary indices in SA are removed and the new array maintain the lexicographical order.

> **Lemma 6.9-1** $SAIS - FLS$ return a new array $SA - FLS$ that is sorted in lexicographical order.

Proof By Contradiction

Let S be a string of strings, where each string is concatenated with the termination symbol \$.

Let SA be the suffix array for string S and let n denote the number of characters in SA. Suppose SA is sorted lexicographical for all suffixes in S.

Suppose that S[SA-FLS[i]] to S[SA-FLS[j]], where i < j < |SA-FLS| is sorted in lexicographical order. Suppose that $S[SA - FLS[j + 1]]$ is lexicographical smaller than $S[SA - IS - FLS[j]]$, that would suggest that SA for S is not sorted lexicographical for all suffixes in S, which is a contradiction. Furthermore, since SA is scanned from left to right and supposed sorted in lexicographical order, each item put in $SA - FLS$ must have been appended in lexicographical order.

> **Lemma 6.9-2** $SAIS - FLS$ return a new suffix array, $SA - FLS$, containing all indices from SA for $S = S_0\$S_1\$...S_{n-1}\$$ where $S[SA - FLS[i] + len] = \$$, $len = |S_0| = |S_1| = \ldots = |S_{n-1}|\$$ and $0 < i < n$.

Proof By Contradiction

Suppose that there exist some i and j, $i < j$, in SA, $0 < i < j < |SA|$ and $len = S_0$ in

$S = S_0$,S_0 =\$, where $S[SA[i] + len] =$ \$ and $S[SA[j] + len] =$ \$. Suppose that SA-FLS contain one item, that would suggest that $i = j$ which is a contradiction.

Suppose that SA is sorted in lexicographically order for all suffixes in $S = S_0\$S_1\$...S_{n-1}\$$ where $S_0, S_1, \ldots, S_{n-1}$ does not contain the termination symbol \$ and $len = |S_0| = |S_1| = \ldots = |S_{n-1}|$.

Suppose that all indices from SA, where $S[SA[i] + len] =$ \$, $0 < i < n$, has been successfully added to the array $SA - FLS$. Suppose that there exists some j in $SA - FLS$ where $S[SA - FLS[j] + len] =$ \$, that would suggest that there exists an index $SA[i] = SA[j]$ where $S[SA[i] + len] =$ \$, but that is a contradiction, since only indices that are bound by $S[SA - FLS[i] + len] =$ \$ was added to $SA - FLS$.

Lemma 6.1-1 and Lemma 6.1-2 suggest that $SA - FLS$ contains indices in lexicographical sorted order and are bound by $S[SA - FLS[i] + len] =$ \$. Furthermore, the length of $SA - FLS$ is proportional to the number of the fixed length distinct string in $S = S_0\$S_1\$...S_{n-1}\$$. For large fixed length strings such as SHA1, SHA256 or MD5 hashes, $SA - FLS$ concededly reduce the number of indices stored. A string consisting of 27.000.000 MD5 hashes would produce a suffix array consisting of 27.000.000 x 33 = 891.000.000 indices, while SA-FLS contains only 27.000.000 indices, which is a reduction factor of 33. For the Sha256, the reduction factor would be 257, hence the length of the hash plus the termination symbol.

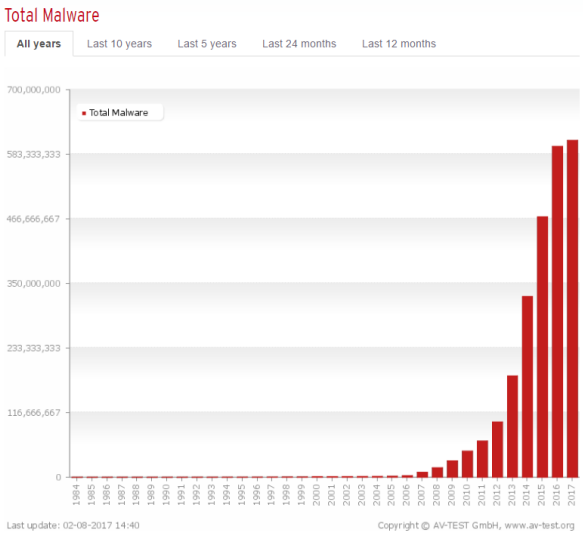


Figure 31: According to AV-TEST, an independent IT-security institute https://www.av-test.org/en/statistics/malware/, 390.000 new malicious programs are identified each day, bringing the total malware count obove 580.000.000 for 2017.

6.10 The importance of space usage& compare with other SACA

We saw examples of preprocessing unstructured text into suffix trees and suffix arrays, examples of searching in $O(n)$ or O(n log m) time complexity and transformed strings that are easier to compress using Burrows-Wheelers Transform. We introduced an algorithm

that could reduce space requirements for fixed length strings with a factor proportional to the length of the fixed length string in S. Space usage is a factor, since sizes of data sets in some applications, as molecular biology, data compression, data mining and text retrieval, to name a few, can be incredibly large. In many cases the alphabet size |E| is typically a fixed constant, such as ASCII E=256 or E= 4 for DNA sequences. In such cases suffix trees and suffix arrays are larger than the text by a multiplicative factor of $O = (log_{\|E\|}n) = O(logn)$. To illustrate this, suppose that we have a DNA sequence of n symbols ($\|E\| = 4$) which we would store on a computer with 2n bits. A suffix array for that DNA sequence would use 4 bytes for each n words or 32 bits, which is 16 times larger than the text itself. The question of space usage is important in both theory and practice, since data sets can be incredibly large, especially in the "Big Data" era of next generation sequencing (SAIS-OPT). Nataliya Timoshevskaya et al. presented in 2014 a characterization and optimization of the SA-IS algorithm for suffix array construction, focusing on irregular memory access, using concepts from Burrow-Wheeler Transform, which achieves a 27% improvement in performance on tested data on the already tuned implementation of SA-IS in the Burrow Wheeler aligner used by the bioinformatics community (SAIS-OPT). Roberto Grossi et al. presented compressed suffix arrays and suffix trees that as a concrete example could compress a typical suffix array of 100 megabytes ASCII file to 30-40 megabytes or less, while the raw suffix array would require 500 megabytes (compressed suffix trees).

7 Malware - Malicious Software

8 Implementation

9 Experimental Results

10 Evaluation and recommendations

11 Discussion

12 Future work

13 Conclussion

14 Literature list and references

15 Appendix

A SAIS Algorithm run

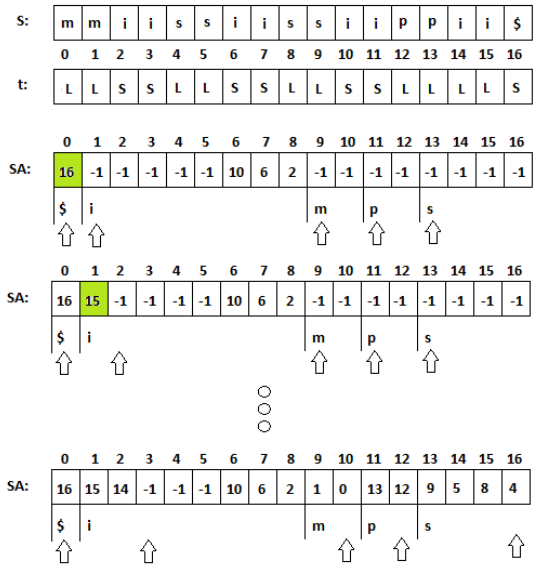


Figure 32: S is scanned from left to right, and indices for each LMS substring is appended to the end of its corresponding bucket in SA. The first LMS substring index is placed at the end of bucket for i, here at position 8 in SA and forwards the bucket end one to the left, hence the bucket end for i now rest at position 7 in SA. This process is repeated until all LMS substring indicies are placed in their buckets.

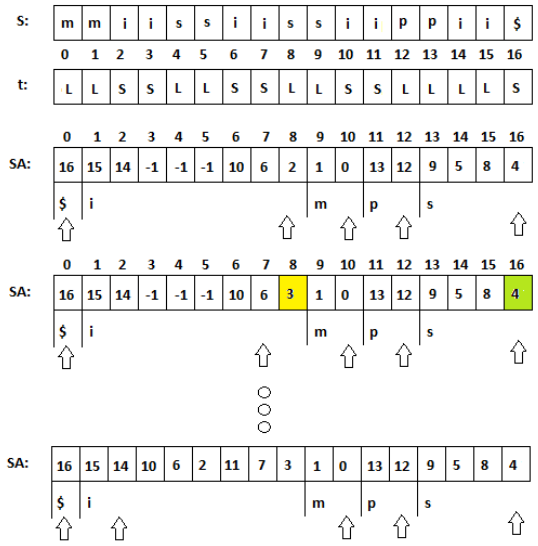


Figure 33: S is scanned from left to right, and indices for each LMS substring is appended to the end of its corresponding bucket in SA. The first LMS substring index is placed at the end of bucket for i, here at position 8 in SA and forwards the bucket end one to the left, hence the bucket end for i now rest at position 7 in SA. This process is repeated until all LMS substring indicies are placed in their buckets.

B SAIS Recurssive step

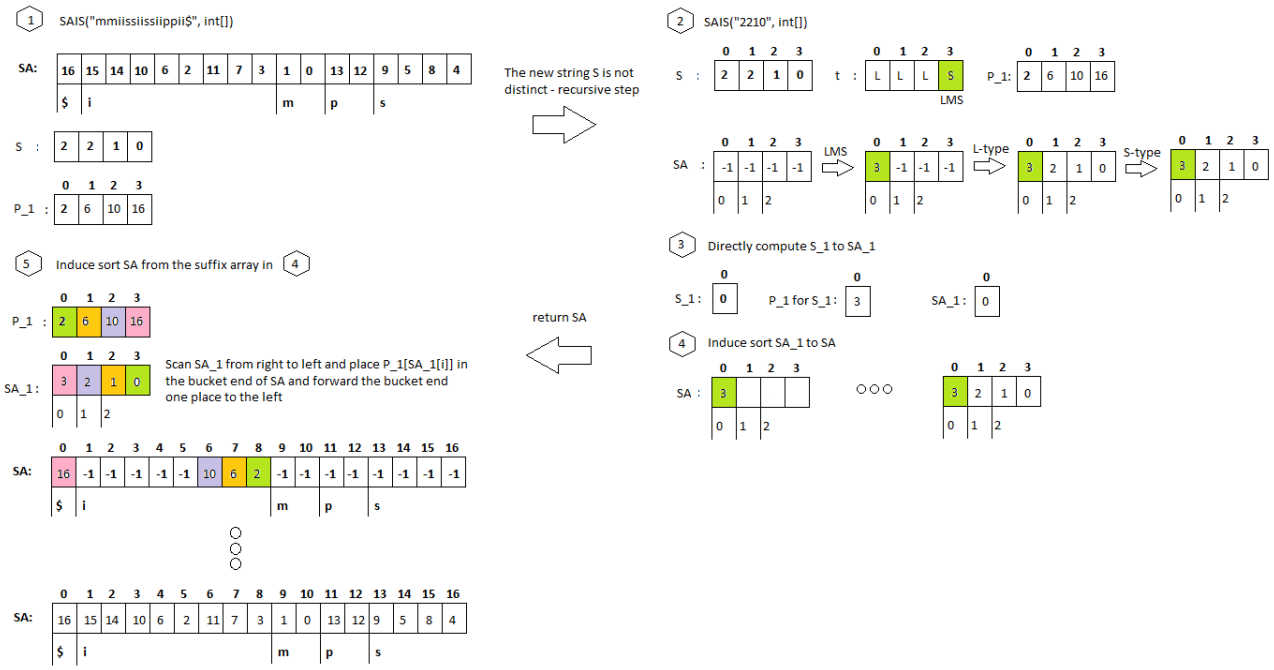


Figure 34: Some description here

C Reversing a compressed string using bwt(S)

References

[1] "Strings - advanced data structures." https://www.youtube.com/watch?v=F3nbY3hIDLQl.

[2] D. Gusfield, *Algorithms on strings, trees, and sequences : computer science and computational biology.* The Pres Syndicate Of The University Of Cambridge, 1 ed., 1997.

[3] R. L. R. . C. S. Thomas H. Cormen, Charles E. Leiserson, *Introduction To Algorithms.* The MIT Pres, Cambridge, Massachusetts, London, England, 3th ed., 2009.

[4] D.-N. L. Nguyen Le Dang and V. T. Le, "A new multiple-pattern matching algorithm for the network intrusion detection system," *IACSIT International Journal of Engineering and Technology*, vol. 8, no. 2, pp. 1–7, 2016.

[5] K. Sadakane, "Compressed suffix trees with full functionality," *2007 Springer Science + Business Media, Inc*, pp. 1–19, 2005.

[6] S. Z. Ge Nong and W. H. Chan, "Two efficient algorithms for linear time suffix array construction," *IEEE Transaction on Computers*, pp. 1–20, 2011.

[7] W. F. S. Simon J. Puglisi and A. Turpin, "The performance of linear time suffix sorting algorithms," *Proceedings of the 2005 Data Compression Conference (DCC'05)*, pp. 1–10, 2005.

[8] U. Baier, "Linear-time suffix sorting – a new approach for suffix array construction," *Institute of Theoretical Computer Science, Ulm University*, pp. 1–10, 2016.

[9] G. Manzini, "An analysis of the burrows-wheeler transform," *Journal of the Association for Computing Machinery*, vol. 48, no. 3, pp. 407–430, 2001.

[10] B. Langmead, "Introduction to the burrows-wheeler transform and fm index," pp. 1–12, 2013.

[11] E. O. Mohamed Ibrahim Abouelhoda, Stefan Kurtz, "Replacing suffix trees with enhanced suffix arrays," *Journal of Discrete Algorithms 2*, pp. 53–86, 2004.

[12] S. A. Toru Kasai, Hiroki Arimura, "Effcient substring traversal with suffx arrays," *DOI Technical Report 185 (2001)*, 2001.

[13] A. L. O. Ferreira, Artur J. and M. A. Figueiredo, "On the suitability of suffix arrays for lempel-ziv data compression.," *International Conference on E-Business and Telecommunications. Springer Berlin Heidelberg*, 2008.

[14] M. Farach, "Optimal suffixx tree construction with large alphabets," *Department of Computer Science, Rutgers University, Piscataway, NJ 08855, USA.*, pp. 1–11, 1997.

[15] "Fast algorithms for finding nearest common ancestors." `http://epubs.siam.org/doi/pdf/10.1137/0213024`. Accessed: 2016-12-20.

[16] "Fast algorithms for finding nearest common ancestorson finding lowest common ancestors: Simplification and parallelization. siam journal on computing, 1988, vol. 17, no. 6 : pp. 1253-1262." `http://epubs.siam.org/doi/abs/10.1137/0217079`. Accessed: 2016-12-20.

[17] E. Ju and C. Wagner, "Personal computer adventure games: Their structure, principles, and applicability for training," *ACM SIGMIS Database*, vol. 28, no. 2, pp. 78–92, 1997.

[18] A. Baltra, "Language learning through computer adventure games," *Simulation and Gaming*, vol. 21, pp. 455–452, December 1990.

[19]

[20] D. M., "How to use scratch for digital storytelling." https://www.graphite.org/blog/how-to-use-scratch-for-digital-storytelling.

[21] https://www.khanacademy.org/computer-programming/new/pjs.

[22] B. Fry and C. Reas. http://processingjs.org/.

[23] L. K. G. at MIT Media Lab, "Scratch." https://scratch.mit.edu/.

[24] J. E. Ormrod, *Educational Psychology: Developing Learners*. Upper Saddle River, N.J.: Pearson/Merrill Prentice Hall, 5th ed., 2006.

[25] "Programming and problem solving (pop)." http://kurser.ku.dk/course/ndab15009u/2015-2016, 2015/2016.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, third ed., 2009.

[27] S. Denmark, "Cultural habits survey 2012." http://www.dst.dk/en/Statistik/dokumentation/declar habits-survey.

[28] J. M. Wing, "Computational thinking and thinking about computing." https://www.cs.cmu.edu/afs/cs/usr/wing/www/talks/ct-and-tc-long.pdf, 2008.

[29] E. Alinea, "iskriv." http://iskriv.dk/, 2012.

[30] D. Statistik, "Kvub1204: Children who play computer games by frequency and background." http://www.statistikbanken.dk/KVUB1204, 2015.

[31] T. May and B. K. Walther, *Computerspillets Fortællinger*, vol. 1. Gyldendal, 2013.

[32] L. Blum and T. J. Cortina, "CS4HS: An Outreach Program for High School CS Teachers," *Sigcse '07*, pp. 19–23, 2007.

[33] S. Gray, C. S. Clair, R. James, and J. Mead, "Suggestions for graduated exposure to programming concepts using fading worked examples," *ICER*, pp. 99–110, 2007.

[34] Y. B. Kafai, "Playing and Making Games for Learning: Instructionist and Constructionist Perspectives for Game Studies," *Games and Culture*, vol. 1, no. 1, pp. 36–40, 2006.

[35] T. Nousiainen, *Children's Involvement in the Design of Game-Based Learning Environments*, pp. 49–66. Springer Science, 2009.

[36] J. Moreno-León and G. Robles, "Computer programming as an educational tool in the English classroom," in *2015 IEEE Global Engineering Education Conference*, pp. 961–966, 2015.

[37] J. M. Wing, "Computational thinking and thinking about computing.," *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, vol. 366, no. 1881, pp. 3717–3725, 2008.

[38] J. M. Wing, "Computational thinking," *Communications of the ACM*, vol. 49, no. 3, pp. 33–35, 2006.