FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

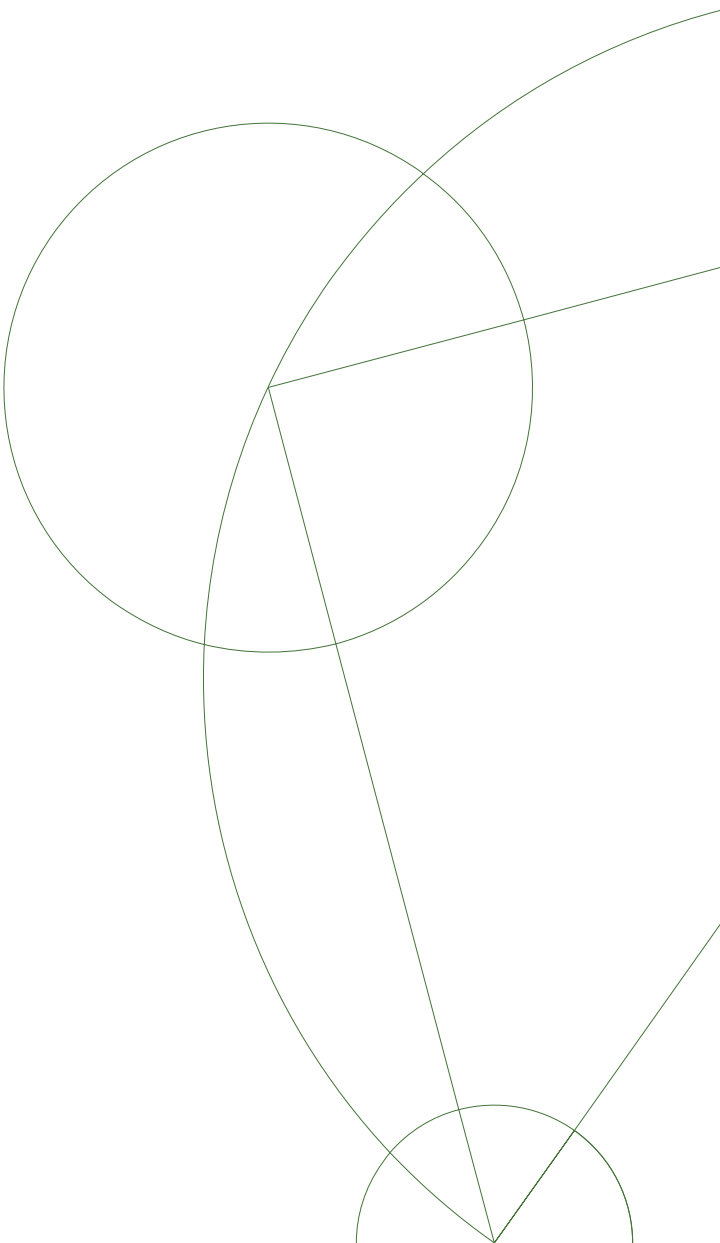
DEPARTMENT OF COMPUTER SCIENCE

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

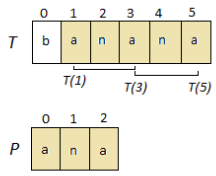


Figure 1: The text T={banana} and pattern P={ana} over the alphabet Σ={abn}. The pattern P occours in T in, at position $T[1]$ and $T[3]$. Notice that occurrences of P may overlap.

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].

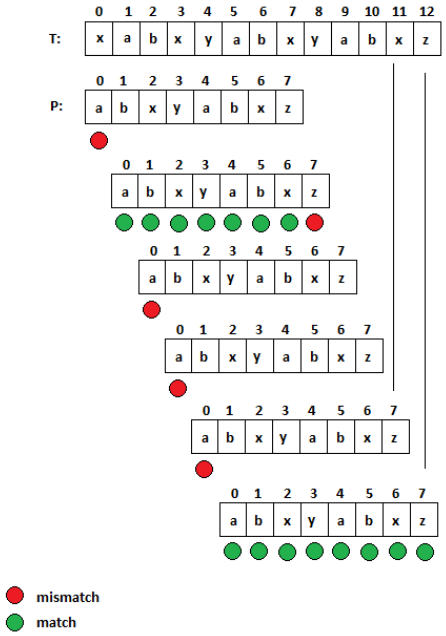


Figure 2: The naive method, where P is shifted one character to the right after each mismatch.

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].

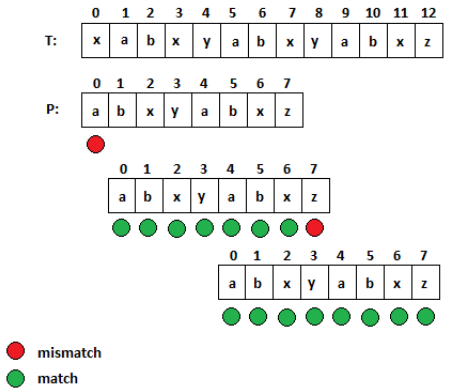


Figure 3: After a mismatch, P is shifted to the next occurrence of a at position 5 in T, moving through T more rapidly

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 $P[0] = a$

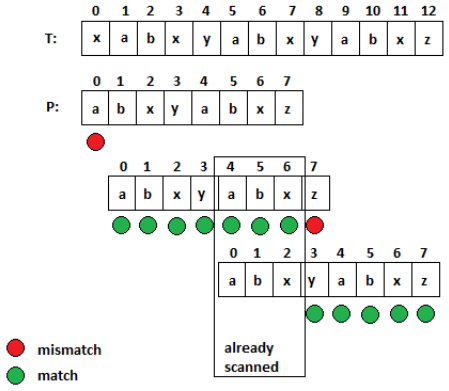


Figure 4: Characters that have already been scanned are stored, so when P is shifted to position 5 in T, abx have already been scanned and can be skipped, and the character scanning is resumed from position 8 and 3, in P and T, respectively.

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].

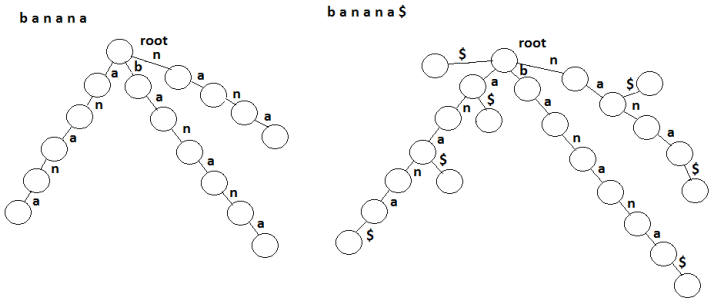


Figure 5: Left is a trie of the string *banana* and the right is a trie of the string *banana*\$.

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.

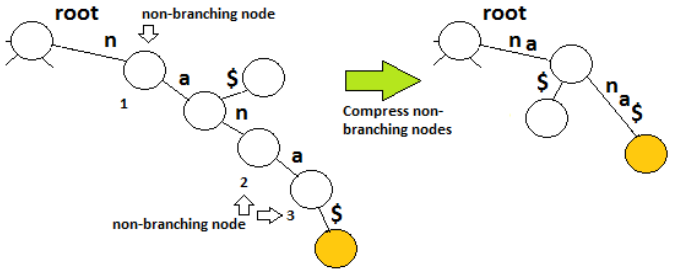


Figure 6: Compressing a trie.

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].

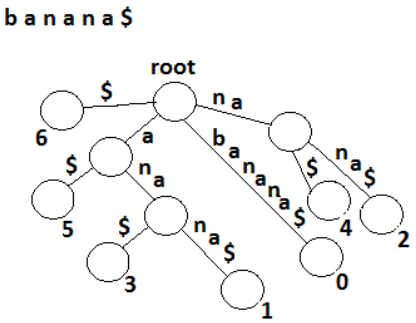


Figure 7: A suffix tree T for string *banana*\$.

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 Operations on suffix trees

Bla bla bla...

Insertion & Deletion Lowest Common Ancestor
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 [6]) and later on simplified by Schieber and Vishkin (1988, published online in 2006 [7])[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, 8]. 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. 8, 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, $lca[2, 8] = 1$.

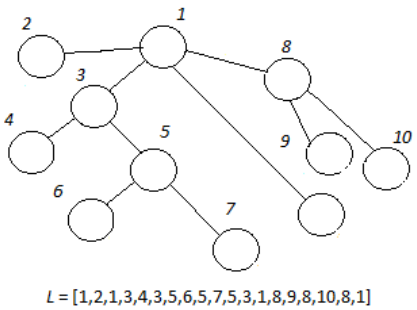


Figure 8: Rooted tree - deept-first travesal with $L = [1, 2, 1, 3, 4, 3, 5, 6, 5, 7, 5, 3, 1, 8, 9, 8, 10, 8, 1]$

Longest Common Prefix

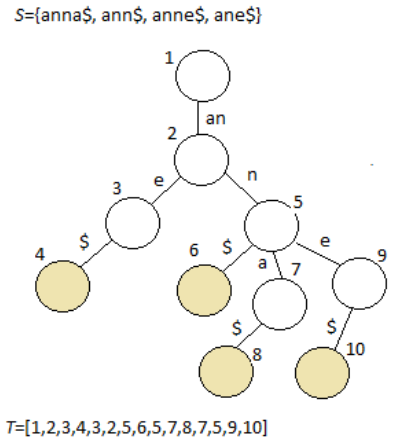


Figure 9: Rooted tree - deept-first travesal with $L = [1, 2, 1, 3, 4, 3, 5, 6, 5, 7, 5, 3, 1, 8, 9, 8, 10, 8, 1]$

What are the usages
Predecessor & Successor Amongst Strings
1 side
Lowest Common Extension
1 side

6.3 Suffix Arrays

Suffix array are space efficient alternatives to suffix trees [2, 9]. 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, 9, 10, 11]. 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 [10] 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 [10, 11]. 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 [11].
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 [11].

2. The algorithm should run fast in practice [11].

3. The algorithm should consume as less extra space in addition to the text and suffix array as possible, where constant amount is optimal [11].

Although no current SACAs fulfill the requirements in an optimal way, research into faster and more space reducing SACAs continued [11]. 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 [11].

6.4 Suffix Trees To Suffix Arrays In Linear Time

6.5 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 [9]. 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 [9].

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 [9].

Let $suf(S, n-1)$ be the termination symbol and of S-type [9].

Let $S[i]$ be S-type or L-type, if suf(S,i) is S-type or L-type, respectively [9].

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 [9].
- $S[i]$ is L-type if $S[i] < S[i+1]$ or $S[i] = S[i+1]$ and $suf(S, i+1)$ is L-type [9].

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, $O(n)$ [9].

	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 10: Type array t is filled from right to left

Figure 10 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 10 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 [9].

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 [9].

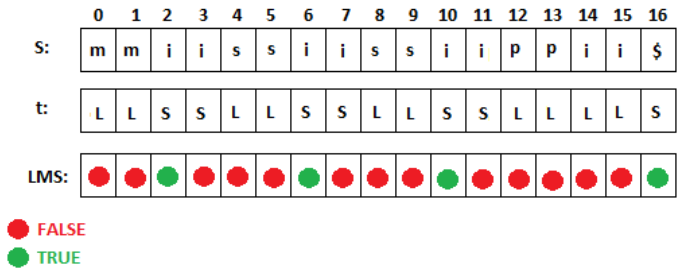


Figure 11: Type array t and LMS array defined for S=mmiissiissiippii$

As Figure 11 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 [9].

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 12 [9]. 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 [9].

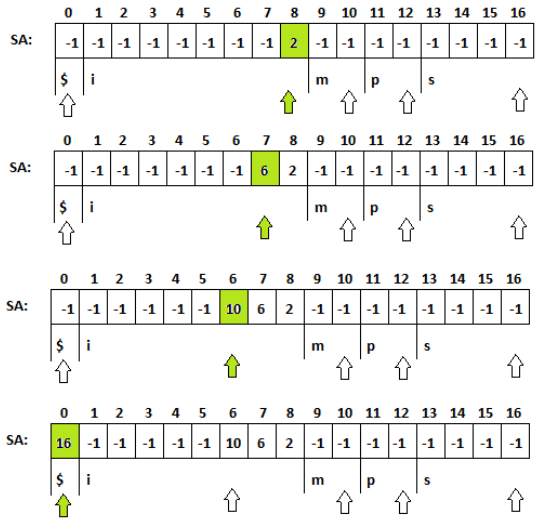


Figure 12: Induced sort of LMS substring

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 [9].

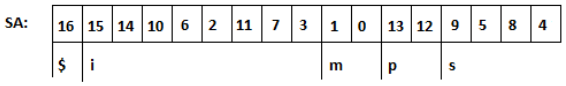


Figure 13: SA after the induced sorting for LMS substring.

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 13 [9].

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 13 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 14. 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 14 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 $P_1 = [2; 6; 10; 16]$ [9].

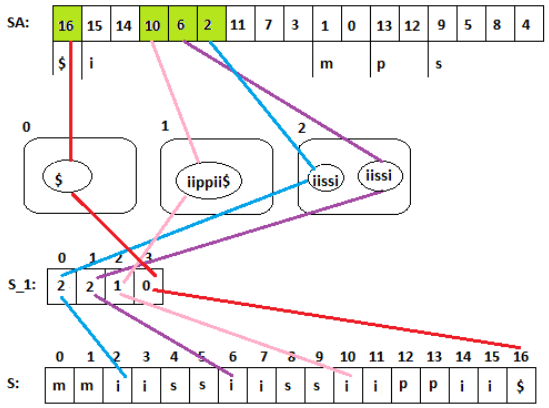


Figure 14: Building S_1 from SA, using the original positions of the LMS substrings in S.

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 15, 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 [9].

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 [9].
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 [9].
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 [9].

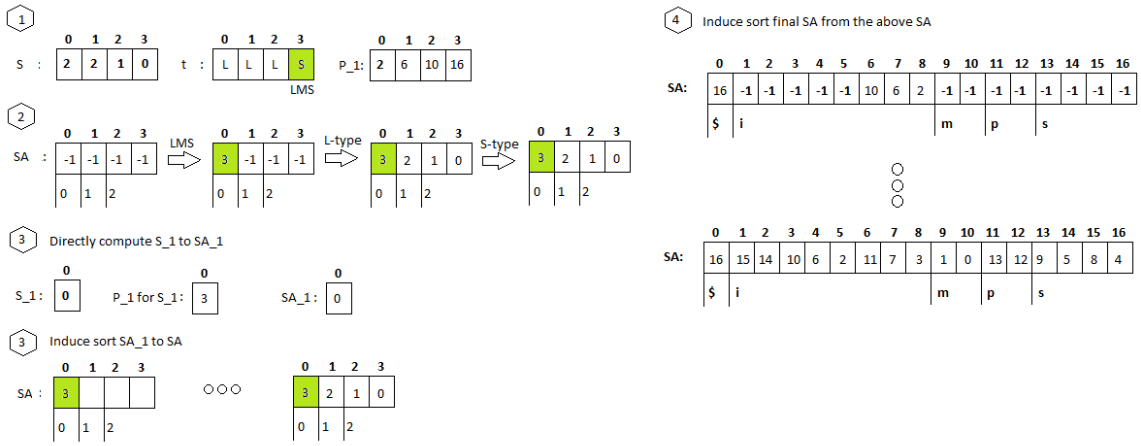
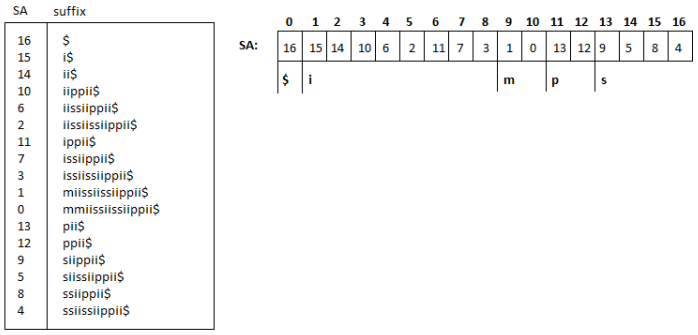


Figure 15: The recursive step $SA(S_1, SA_1)$. First S-types, L-types and LMS characters/substrings for S_1 is localized. Secondly, induced sort LMS substrings, such that $SA_1 = [3; 2; 1; 0]$. Last, scan SA_1 from right to left and place $P_1[SA_1[i]]$ into the buckets end of SA as described for induced sorting LMS Substrings, for each item in SA_1. The final result is displayed in step 4.

This procedure is demonstrated in Figure 21 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 16 [9].
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.



SA	suffix
16	$
15	i$
14	ii$
10	iippii$
6	iissiippii$
2	iissiissiippii$
11	ippii$
7	issiippii$
3	issiissiippii$
1	miissiissiippii$
0	mmiissiissiippii$
13	pii$
12	ppii$
9	siippii$
5	siissiippii$
8	ssiippii$
4	ssiissiippii$

Figure 16: The suffix array for the text T=mmiissiissiippii\$.

6.6 SAIS - Analysis, Correctness & Completeness

5-8 sider

6.7 SAIS - Linear Time Preprocessing

2 sider

6.8 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 17.

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 17: 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_DS 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 $O(N_DS)$ 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
```

Describe the algorithm
Some description here

Analyzing the algorithm - something with loopinvariant
Initialization
Some description here

Maintenance
Some description here

Termination
Some description here

Correctness

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.

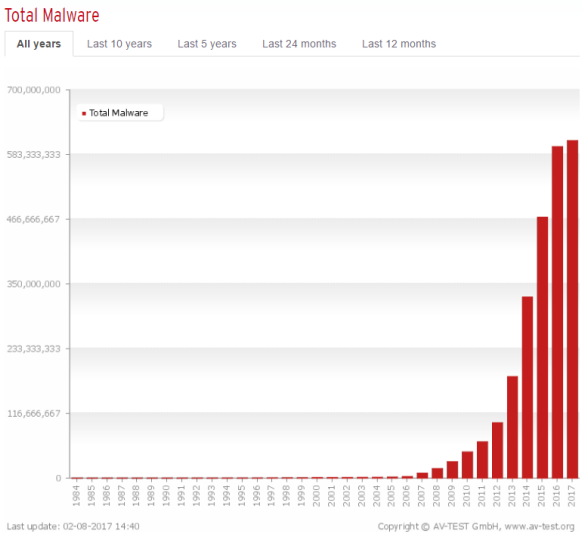


Figure 18: According to AV-TEST, an independent IT-security institute https://www.av-test.org/en/statistics/malware/, 390.000 new malicious programs are identified each day, bringing the total malware count obove 580.000.000 for 2017.

6.9 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 [12].
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 [12].

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 [12]. 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 [12].

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.

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.

6.10 SAIS-OPT - Optimization of the SA-IS Algorithm using Burrows-Wheeler Transform

2-3 sider

7 Malware - Malicious Software

- Encryption - Do md5 ganrantee uniqueness?

A SAIS Algorithm run

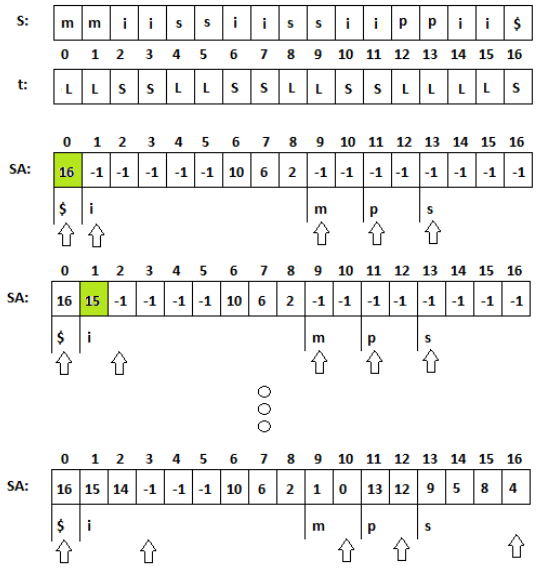


Figure 19: S is scanned from left to right, and indices for each LMS substring is appended to the end of its corresponding bucket in SA. The first LMS substring index is placed at the end of bucket for i, here at position 8 in SA and forwards the bucket end one to the left, hence the bucket end for i now rest at position 7 in SA. This process is repeated until all LMS substring indicies are placed in their buckets.

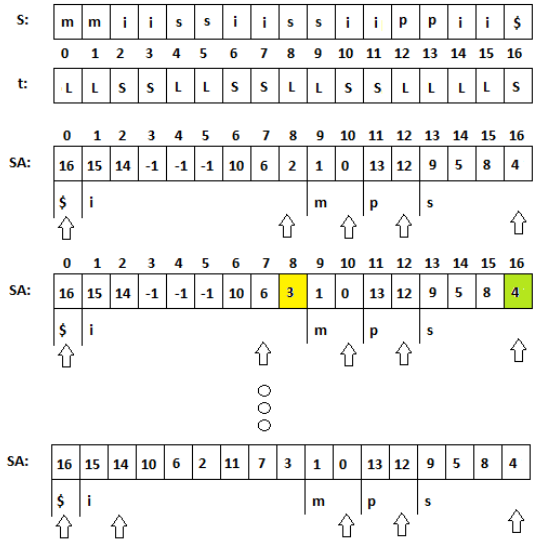


Figure 20: S is scanned from left to right, and indices for each LMS substring is appended to the end of its corresponding bucket in SA. The first LMS substring index is placed at the end of bucket for i, here at position 8 in SA and forwards the bucket end one to the left, hence the bucket end for i now rest at position 7 in SA. This process is repeated until all LMS substring indicies are placed in their buckets.

Etiam pede massa, dapibus vitae, rhoncus in, placerat posuere, odio. Vestibulum luctus commodo lacus. Morbi lacus dui, tempor sed, euismod eget, condimentum at, tortor. Phasellus aliquet odio ac lacus tempor faucibus. Praesent sed sem. Praesent iaculis. Cras rhoncus tellus sed justo ullamcorper sagittis. Donec quis orci. Sed ut tortor quis tellus euismod tincidunt. Suspendisse congue nisl eu elit. Aliquam tortor diam, tempus id, tristique eget, sodales vel, nulla. Praesent tellus mi, condimentum sed, viverra at, consectetuer quis, lectus. In auctor vehicula orci. Sed pede sapien, euismod in, suscipit in, pharetra placerat, metus. Vivamus commodo dui non odio. Donec et felis.

B SAIS Recurssive step

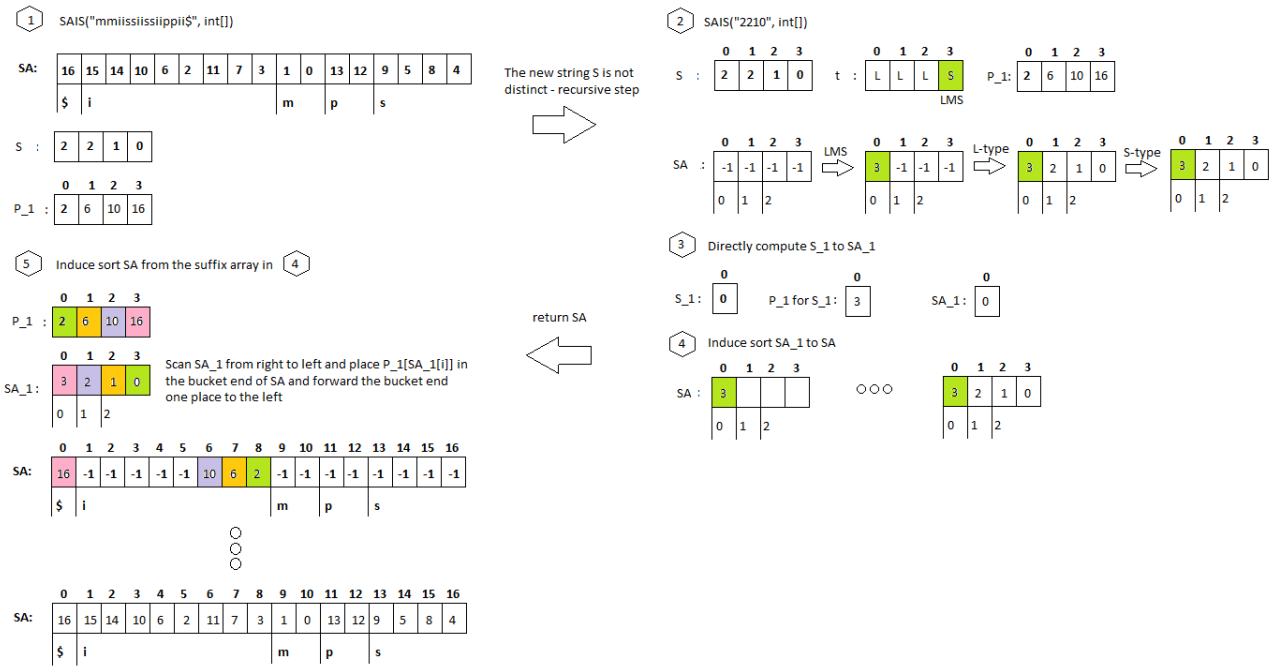


Figure 21: Some description here

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

[7] "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.

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

[9] 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.

[10] 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.

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

[12] G. Manzini, "A new multiple-pattern matching algorithm for the network intrusion detection system," *Journal of the Association for Computing Machinery*, vol. 48, no. 3, pp. 407–430, 2001.

[13] 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.

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

[15]

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

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

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

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

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

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

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

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

[24] 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.

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

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

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

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

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

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