Linear-time Suffix Array Construction by Induced Sorting, SAIS algorithm

Definition of a suffix. Given a text T[0…n] over alphabet Σ of size σ, where the last character $ of T is lexicographically smaller than any other character and occurs in T only once. A ***suffix*** T_i of T is a substring of T[i…n].

Definition. A suffix T_i of T is called L-type if it is lexicographically greater than the suffix T_{i+1}, we denote this relation as $T_i > T_{i+1}$, and S-type if $T_i < T_{i+1}$ (T_i is lexicographically smaller than T_{i+1}). A suffix T_i is called leftmost S-type, LMS, if it is S-type, and suffix T_{i-1} is L-type.

Definition. A substring T[i…j] of T is called LMS-Substring, or in short T* substring, if T_i and T_j are LMS-suffixes, and none of the suffixes starting between *i* and *j* are LMS-suffix. We categorize a character T[i] of T as L-type, S-type and LMS-type according to the category of T_i suffix.

Definition. A suffix array of T, SA, is an array SA[0…n] such that SA[i] = p, where p is a position of suffix T_p and *i* is the lexicographic order of T_p among all suffixes of T. In other words, SA stores the positions of suffixes of T sorted in lexicographic (alphabetic) order.

SA-IS Algorithm Outline [1]
Input: Text T[0…n]
Output: Suffix Array of T, SA
0. Assign type of each suffix in T (L-type, S-type).
1. Induce-sort all LMS-substrings of T in O(n) time.
2. Give each LMS-substring of T a name and construct a shortened string T_1 whose alphabet consists of the names of LMS-substrings. This step takes O(n) time.
3. Call recursively SA-IS on T_1 to calculate the suffix array SA_1 for T_1.
4. Induce SA from SA_1 (i.e. using SA_1 for T_1, construct SA for T) in O(n) time.

Time analysis of SA-IS:
The length of T_1 is at most n/2 since each character of T_1 corresponds to a T* substring in T, and there are at most n/2 LMS-substrings in T (each LMS-substring starts with S-type character and the previous character in T is L-type character by definition). Thus the size of the original problem reduces in half with each recursive call. Time required to execute this algorithm is T(n) = T(n/2) + O(n), which is O(n).

Now we will describe how to implement each step of the algorithm.

We will use an example to illustrate all the steps of the algorithm. In our example,
T = "`m m i s s i s s i i p p i i $`"

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	m	m	i	s	s	i	s	s	i	i	p	p	i	i	$

Maintain an array SA[0…n] (for suffix array), and initially SA[i] = -1, for all $0 \leq i \leq n$. Since array SA store positions of suffixes sorted in alphabetical order, suffixes starting with the same character span consecutive block of suffixes in SA (hereafter, we'll call a block of suffixes starting with the same character a ***bucket***).

We'll need arrays A and C each of size equal to alphabet size σ. Array A at index *c* holds the total number of characters *c* in T. For example, in our example, A[m] = 2 and A[i] = 6. A can be constructed by scanning T once and counting the occurrence of each character c in T. Array C at index *c* holds the starting index of the *c*-bucket in SA. $C[c] = \sum_{i<c} Count(i)$, where Count(i) is the number of all characters i < c in T. In other words, *C*[c] is the total number of characters in T alphabetically smaller than character *c.* In our example, C[i] = 1 (only one character $ is smaller than i, so i-bucket will start at index 1 in SA) and C[m] = 7 (there is one character $ and six characters *i* smaller than *m*, so m-bucket in SA starts at index 7).

For example, let character $ be index 0, *i* be index 1, *m* be 2, *p* denote index 3 and *s* denote index 4 in A. Then after counting the number of occurrences of each character in T, A is

A =

Index of character *c*	$0_\$$	1_i	2_m	3_p	4_s
Occurrences of *c* in T	1	6	2	2	4

Then we scan A from index *j* = 1, 2, 3, 4 and calculate Count(*c*), the total number of characters *i* in T that are lexicographically smaller than *c*, updating the values stored in C. C[0] = 0 (no characters are smaller than character $ indexed by 0). C[c] = C[c-1] + A[c-1].

C =

Index of character *c*	$0_\$$	1_i	2_m	3_p	4_s
$\sum_{i<c} Count(i)$	0	1	7	9	11

We need to maintain the pointers to heads (or ends) of c-buckets (a bucket starting with c-character) during the algorithm. So, we will keep another array B of size σ such that B[c] = Head(c), where Head(c) is the index of SA of the current head of the c-bucket. For Step 0, we need the information about current tails of c-buckets, i.e. pointers are placed at the **End** of each c-bucket.
So, initially, B[c] keeps the index of the end of each c-bucket: B[s] = n, and B[c] = B[c+1] – A[c+1].

B =

Index of character *c*	$0_\$$	1_i	2_m	3_p	4_s
End of c-bucket	0	6	8	10	14

Step 0. *Assign type of each suffix in T (L-type, S-type).* For this step, we'll need a bit-array (or bool array) *t,* such that t[i] = 0 if T_i is L-type suffix and t[i] = 1 if T_i is S-type suffix. Scan T from **Right-to-Left**. Initialize $ as S-type, then if T[i] > T[i+1] then t[i] = 0 (L-type), and if T[i] < T[i+1] then t[i] = 1 (S-type). Finally if T[i] = T[i+1], then check t[i+1], if it is L-type, then t[i] is also L-type; otherwise, t[i] is S-type (see **Proof 4**).

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

At the same time as assigning L-type to T[i], check if T[i+1] is S-type, then T[i+1] is LMS, so store it's position (i+1) into SA at the end of the T[i+1]-character bucket:
SA[B[T[i+1]]] = i + 1

For example, as we scan T[13], we define it as L-type, check that t[14] is S-type, and this means that T[14] is LMS, so we need to put position 14 at the end of the $-bucket (where $ is the character at position 14 in T). We find from B[$] that the end of this bucket is index 0 of SA, so put 14 into SA[0] and update the current end of bucket by decrementing it by 1, i.e. B[T[i+1]] = B[T[i+1]] - 1. Now B[$] = 0-1 = -1.
As we scan L-type T[7], t[8] is S-type, meaning T[8] is LMS, and T[8] is character *i*, and the current end of the i-bucket is B[*i*] = 6, so place 8 into SA[6] and update the end of i-th bucket by decrementing 6 by 1, i.e. B[*i*] = 6 – 1 = 5.
After scanning T from right to left, we stored starting positions of all LMS-substrings at the end of corresponding c-buckets (where c is the starting character of an LMS-substring).

c-bucket	$	*i*						*m*		*p*		*s*			
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA	14				2	5	8								

Step 1. Sort all LMS-substrings of T in O(n) time. At the beginning of this step all LMS-substrings are placed in their corresponding buckets in SA (by Step 0).

a) Recalculate B array so that each B[c] stores the **Head** of the c-bucket, i.e. B[c] = C[c].

Index of character *c*	$	i	m	p	s
$B[c]$	0	1	7	9	11

b) Induce-sort L-type suffixes using LMS-substrings, i.e. fill in SA[i] entries such that SA[i] = p and p is the starting position of an L-type suffix.

Scan SA from **Left-to-Right** and for each p = SA[i] (such that p is not equal to -1), if T[p-1] is L-type (check if t[p-1] = 0), then store p-1 at the current head of the T[p-1]-bucket: SA[B[T[p-1]]] = p-1;
and update B[T[p-1]] by incrementing its value by 1, i.e. B[T[p-1]] = B[T[p-1]] + 1.

For example, starting at SA[0], p = SA[0] = 14, and t[14-1] = 0 (T[13] is L-type), so place 14-1 = 13 into the current head of T[13] = i bucket. The head is located at B[i] = 1, so set SA[1] = 13. Update B[i] = B[i] + 1 = 1 + 1 = 2.

c-bucket	$	*i*						*m*		*p*		*s*			
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA	14	13	12		2	5	8								

Next, SA[1] = 13, and t[13-1] is L-type, so find the current head of T[12] = i bucket, B[i] = 2, and store 12 at SA[2]. Update B[i] = 2 + 1 = 3.

At the end of scanning of SA from left to right, SA is:

c-bucket	$	*i*						*m*		*p*		*s*			
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA	14	13	12		2	5	8	1	0	11	10	4	7	3	6

c) Reset the values of B to point to the **End** of c-buckets. B[s] = 14, and B[c] = B[c+1] – A[c+1].

d) Induce the order of S-type suffixes from ordered L-type suffixes, i.e. fill in entry SA[i] such that SA[i] = p and p is the starting position of S-type suffix.

Scan SA from **Right-to-Left**, and for each SA[i] = p (valid, not -1), if T[p-1] is S-type suffix (t[p-1] = 1), then store p-1 at the end of the T[p-1]-bucket:
SA[B[T[p-1]]] = p-1;
and update the end of the bucket by moving it one to the left (decrement by 1): B[T[p-1]] = B[T[p-1]] – 1.

NOTE (for calculations of p-1 and p-2, previous two positions). While scanning SA, if p = SA[i], and you need to calculate (p-1) or (p-2), there might occur out-of-range issue that you need to address appropriately. If SA[i] = p = 0, then p-1 is out-of-range index, use previous position to be equal to T.size – 1 (the last idex) instead of p – 1. Previous position of position 0, is T.size – 1 (in our example it is 14), and instead of p – 2, use T.size – 2. Similarly, if position p = 1, then p-2 is out-of-range, so use T.size – 1 for this case (think of T as a circle starting at 0 and ending with T.size-1, so previous index of 0 on the circle is T.size-1).

For example, SA[14] = 6, and T[6-1] = T[5] is S-type suffix, so place 5 at the end of T[5] = i bucket, which corresponds to index 6 of SA (note, SA[6] is over-written and now SA[6] = 5). Update B[i] = 6 – 1 = 5. Now the end of i-bucket is index 5 of SA.

c-bucket	$	*i*						*m*		*p*		*s*			
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

SA	14	13	12		2	5	**5**	1	0	11	10	4	7	3	6

Next, SA[13] = 3, and T[3-1] = T[2] is S-type suffix, so place 2 at the end of T[2]=i bucket, which corresponds to index 5 of SA (thus, SA[5] = 2). Update the end of the i-bucket by decrementing it by one: B[i] = 5 – 1 = 4.

c-bucket	$	*i*						*m*		*p*		*s*			
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA	14	13	12		2	**2**	**5**	1	0	11	10	4	7	3	6

At the end of scanning from right to left, all LMS-substrings are correctly sorted relative to each other, but the order of S-type suffixes or L-type suffixes are not necessarily correct.

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

c-bucket	$	*i*						*m*		*p*		*s*			
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA	14	13	12	**8**	**9**	**2**	**5**	1	0	11	10	4	7	3	6
L	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0

At this step, we also maintain a bit-array (or a bool array) L such that L[i] = 1 if SA[i] = p corresponds to LMS-substring starting at p, and L[i] = 0, otherwise. While scanning SA[i] = p and checking whether T[p-1] is of S-type, if so, then we also need to check whether T[p-2] is of L-type, then T[p-1] is LMS-substring, so set L[B[T[p-1]] = 1.

At the end of Step 1, all LMS-substrings are correctly sorted relative to each other.

Step 2. Give each LMS-substring of T a name and construct a shortened string T_1 whose alphabet consists of the integer-names of LMS-substrings.

Idea is to scan SA from **Left-to-Right** and assign LMS-substrings names, integer-ranks of the LMS order in SA. For example if T_i is LMS-substring that occurs before another LMS-substring T_j in SA, then T_i's name will be an integer less than T_j's name if both substrings are different, or the same integer if the substrings are the same (same corresponding characters and same type characters, L-type character are smaller than S-type). Only two consecutive LMS-substrings in SA are needed to be compared (they are stored in alphabetic order, and only consecutive substrings may be the same). Write a function that compares two LMS-substrings (pass T by constant reference, pass array t, and two LMS substrings are passed each by an integer, starting position of the LMS in T); the function returns *true* if the two LMS substrings at position i and position j are same (same characters and same character-types) and returns *false* if the LMS substrings are different. Note that we can use information stored at array L to check if index i corresponds to p = SA[i] such that L[i] = 1 (LMS starts at position p in T), but we don't know the length of an LMS. We can use a definition of LMS: the first and the last characters of LMS are LMS-characters and no other characters between the first and the last are LMS-characters; so as we scan t array we can keep a track of whether the current character of either LMS substrings is LMS character, and if so, stop scanning (the end of one of LMS-substrings is reached).

The new shortened string T_1 is formed from the names (assigned integers) of LMS substrings preserving the order of occurrence of LMS-substrings in the original string T.

<u>To clarify, the names are assigned 0, 1, 2, …, to the LMS-substrings in the order of their occurrence in SA, but these names are put into a new string in the order of the corresponding LMS-substrings in the original string T.</u>

We'll keep T_1 as an array of integers of size equal to the total number of LMS-substrings in T, call this number ***x*** (T_1 has alphabet {0, 1, …, k}, where (k+1) is the size of the alphabet equal to the number of distinct LMS-substrings).

Now we'll explain how to implement this step. We need a temporarily pointer (variable) to one LMS-substring, that was processed immediately before LMS-substring that is being currently processed (call this pointer *previos*). Initially, *previous* is set to SA[0]:

previous = SA[0]

(a) The first LMS-substring corresponds to the special $ character (sentinel), and it's name is 0 (no other LMS is identical to this one consisting of a single character). Set T_1[x-1] = 0 (the last character of T_1 is 0, the smallest in the alphabet).

(b) **Fill in N.** Maintain an array N (names of LMS substrings) of the same size as size of T (initialize N[i] = -1 for all i = 0, 1,…,n). N[n] = 0, (corresponds to the last character of the string $).
Scan SA from **Left-to-Right**, and for each SA[i], i = 1…n, check if L[i] = 1. If so, then LMS-substring occurs at position p = SA[i] in T, and this is <u>*currently*</u> processed LMS-substring. To assign a name to this LMS-substring, compare this current LMS to the LMS pointed by *previous* pointer. If these LMS substrings are identical, then the current LMS substring is assigned the same name as the previous LMS-substring; otherwise, increment the name of the previous LMS-substring by one and assign this integer to the current LMS-substring. Once we assigned a name ***m*** to an LMS-substring, we know that it occurs at position *p*=SA[i] in T, so set N[p] = N[SA[i]] = m. Set *previous* to SA[i].

(c) **Fill in T_1.** Do this step only after array N is filled in (step b). To build T_1, scan array N from **Left-to-Right** and fill in T_1 from **Left-to-Right**:
T_1[j] = N[p] (where N[p] is not -1)
Note, that we need to use two different variables *j* and *p* to access T_1 and N (because T_1 and N are of different lengths).

Example on the execution of Step 2.

c-bucket	\$	*i*						*m*		*p*		*s*			
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA	14	13	12	**8**	**9**	**2**	**5**	1	0	11	10	4	7	3	6
L	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0

While scanning SA from **Left-to-Right**, for i = 1,…,n (i = 0 is processed already), we discover that SA[3] corresponds to LMS-substring. Compare it to the previous LMS pointed by *previous* = SA[0] = 14. The current LMS-substring starting at position SA[3] = 8 in T is "iippii\$", and the previous LMS-substring starting at position 14 in T is "\$". The LMS-substrings are different, so the name for "iippii\$" is 1 (one greater than the previous name 0). Set N[SA[3]] = N[8] = 1 (the name of the LMS substring is 1, LMS substring starts at position 8 in T).

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	m	m	i	s	s	i	s	s	i	i	p	p	i	i	\$
t	L	L	S	L	L	S	L	L	S	S	L	L	L	L	S
N									1						0

The next LMS-substring is found at SA[5], corresponding to position SA[5] = 2 in T, and the content of the current LMS-substring is now "issi", which is not identical to the previous LMS-substring "iippii\$", so the name of the "issi" is 2 (one greater than the name of the previous LMS-substring, which was 1). Set N[SA[5]] = N[2] = 2.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	m	m	i	s	s	i	s	s	i	i	p	p	i	i	\$
t	L	L	S	L	L	S	L	L	S	S	L	L	L	L	S
N			2						1						0

The next LMS-substring is found at SA[6] and it starts at position 5 in T corresponding to "issi", it is identical to the previous LMS-substring (that started at position 2 of T), so the name of the current LMS-substring is the same as the name of the previous LMS-substring, namely 2. Set N[SA[6]] = N[5] = 2:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	m	m	i	s	s	i	s	s	i	i	p	p	i	i	\$
t	L	L	S	L	L	S	L	L	S	S	L	L	L	L	S
N			2			2			1						0

Not filled entries in array N are initialized to -1. There are no more LMS-substrings. So now, we are ready to scan array N and fill in array T_1:

Index of T_1	0	1	2	3
T_1	2	2	1	0

T_1 is now "2210". Make a recursive call on T_1.

Step 3. Call recursively SA-IS on T_1 to calculate the suffix array SA_1 for T_1.

Termination step: check if all characters of T_1 are different (if the size of the alphabet is smaller than the length of T_1, then some characters are repeated).

If all characters are different, we can calculate SA_1 by placing each character into it's proper bucket in SA_1 and terminate the recursive call. For this, scan T1 from **Left-to-Right** using index i = 0, 1, 2…, and do:
SA1[T1[i]] = i;

Step 4. **After Recursive Call:** Induce SA from SA_1 (i.e. using SA_1 for T_1, construct SA for T) in O(n) time.

The idea is to place sorted LMS-substrings in the same relative order from SA1 to SA, and then induce-sort L-type suffixes from the order of LMS-substrings, and finally, induce-sort S-type suffixes from the order of L-type suffixes.

(a) Reset B so that B[c] points to the **End** of *c*-bucket.

(b) Reset SA[i] = -1, for all i = 0, 1, …, n.

(c) Recall that i-th character in T_1 corresponds to i-th LMS-substring in the original string T.
SA_1[j] = i gives us the i-th suffix of T_1 that corresponds to i-th LMS-substring in T.

For this step, given i-th suffix of T_1 (hence, i-th LMS in T), we need to retrieve the starting position of i-th LMS in T. Since we do not need to use T_1 anymore, we may overwrite it's i-th integer (corresponding to i-th LMS in T) with the starting position of i-th LMS in T. Scan *t* array (that holds L- and S-types) Left-to-Right using index j, and at the same time fill in T_1 using index i. If t[j] is of S-type and t[j-1] is of L-type, then T[j] corresponds to LMS-substring, so set T_1[i] = j.

Now we can place positions of LMS-substrings into SA. Scan SA1 from **Right-to-Left** for j = x-1, …, 0 (where x is the total number of LMS-substrings in T). We scan from right to left, because we want to put positions of LMS-substrings into SA at the end of the corresponding c-buckets (so all LMS-substrings are at the end of c-buckets in the preserved order of their occurrence in SA1).

For each SA_1[j] = i, find *p* = T_1[i], which is the starting position of the i-th LMS-substring in T. Then T[p] is the character *c*, so we know in which *c*-bucket to place position *p* into SA.
SA[B[T[p]]] = p;
B[T[p]] = B[T[p]] – 1; decrement by 1 to point to the current not occupied end of the bucket.

Example of execution of Step 4.

Let SA_1 of T_1 be:

Index	0	1	2	3
T_1	2	2	1	0
SA_1	**3**	**2**	**1**	**0**

Overwrite T1 with the starting positions of LMS-substrings in T using t-array:

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

Index	0	1	2	3
T_1	2	5	8	14
SA_1	**3**	**2**	**1**	**0**

By scanning SA_1 from **Right-to-Left**, we find $SA_1[3] = 0$, which corresponds to position 0 in T_1, and, hence, to the 0^{th} LMS substring in T. Find position $p = T_1[0] = 2$. Find character $T[p] = T[2] = i$. Place position $p = 2$ into the end of *i*-bucket in SA, i.e. SA[6] = 2.

c-bucket	$	*i*						*m*		*p*		*s*			
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA							2								

Next, $SA_1[2] = 1$, which corresponds to position 1 in T_1, and to 1^{st} LMS substring in T (count of LMS-substrings starts with 0). Find position $p = T_1[1] = 5$. Character T[5] = i, so place position 5 into the current end of the *i*-bucket.

c-bucket	$	*i*						*m*		*p*		*s*			
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA						5	2								

Next, $SA_1[1] = 2$, which corresponds to position 2 in T_1. Find $p = T_1[2] = 8$, and T[8] = i. Place 8 at the end of the i-bucket in SA.

c-bucket	$	*i*						*m*		*p*		*s*			
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA					8	5	2								

Finally, $SA_1[0] = 3$, corresponding to position 3 in T_1. Find $p = T_1[3] = 14$. T[14] = $, so place 14 into the end of $-bucket in SA:

c-bucket	$	*i*						*m*		*p*		*s*			
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA	14				8	5	2								

Now, all LMS-substrings's positions are placed into SA in correct relative order as they appear in SA_1.

Repeat induce-sort of L-type and S-type suffixes that we described in detail in Step 1 to obtain the final SA of T.

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

Reset B[c] to point to the **Head** of *c-buket.*
B[c] = C[c].
Induce sort L-type suffixes:

c-bucket	$	*i*						*m*		*p*		*s*			
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA	14	13	12		8	5	2	1	0	11	10	7	4	6	3

Reset B[c] to point to the **End** of *c-buket.*
B[s] = n, and B[c] = B[c+1] – A[c+1].
Induce sort S-type suffixes:

c-bucket	$	*i*						*m*		*p*		*s*			
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA	**14**	13	12	**8**	**9**	**5**	**2**	1	0	11	10	7	4	6	3

Return SA.

Reference:

Ge Nong, Sen Zhang, and Wai Hong Chan, “Linear Suffix Array Construction by Almost Pure Induced-Sorting”, *Data Compression Conference*, 2009, DCC '09, 16-18 March 2009, DOI: 10.1109/DCC.2009.42