Linear-time suffix sorting with SA-IS

Jesper Larsson

March 1, 2024

This text is about suffix sorting using the SA-IS algorithm of Nong, Zhang, and Chan. It explains how and why the algorithm works, and provides an implementation in Javascript.

1 Problem definition

By suffix sorting, we mean the problem of creating a *suffix array* for a string T of n characters from an alphabet Σ. We write:

- $T[i]$ for an individual character of T, where $0 \leq i < n$,
- $T[i...j]$ for a substring "$T[i]\,T[i+1]\cdots T[j]$", where $0 \leq i \leq j < n$, and
- $T[i...]$ for a suffix $T[i...n-1]$, where $0 \leq i < n$.

The suffix array is a permutation of the integers 0 to $n-1$ in which i comes before j if $T[i...] < T[j...]$ when strings are compared lexicographically.

We assume that characters are integers $0...|\Sigma|-1$, where 0 is a *sentinel*, required to appear at the end of T and only at the end. A valid T can be created from an input string by adding one to every character code and appending a zero sentinel like this:

```
const n = input.length+1;
const T = Array(n);
for (let i = 0; i < n-1; i++) { T[i] = input.charCodeAt(i)+1 }
T[n-1] = 0;
```

For instance, let $T =$ "beebybeeybybaby$" where "$" is the sentinel (zero), and letters correspond to positive integers in the order of the alphabet. Then the suffix array is $(15, 12, 11, 0, 5, 13, 9, 3, 2, 1, 6, 7, 14, 10, 4, 8)$ reflecting the lexicographic suffix order:

```
15: $
12: aby$
11: baby$
 0: beebybeeybybaby$
 5: beeybybaby$
13: by$
 9: bybaby$
 3: bybeeybybaby$
```

```
 2: ebybeeybybaby$
 1: eebybeeybybaby$
 6: eeybybaby$
 7: eybybaby$
14: y$
10: ybaby$
 4: ybeeybybaby$
 8: ybybaby$
```

2 Recursive suffix sorting

Several recursive *divide-and-conquer* algorithms exist for sorting T in time $O(n)$, under the assumption that $|\Sigma|$ is $O(n)$, without first constructing a suffix tree. The general scheme is:

1. Pick out some positions of T to create a string T' such that $|T'|/|T|$ is (at most) a constant $d < 1$.
2. Recursively suffix sort T'.
3. Induce the order of all the suffixes of T from the suffix order of T'.

By making sure that steps 1 and 3 take altogether at most $k \cdot n$ time for some constant k, we get a total time complexity of $O(n)$, since the recursion formula $time(n) \leq kn + k' + time(dn)$ resolves to $time(n) \leq (k/(1-d)) \cdot n + k''$ with constants k' and k''.

The algorithm by Kärkkäinen, Sanders, and Burkhardt has $d = 2/3$ and is reasonably simple. The algorithm by Kim, Sim, Park, and Park has $d = 1/2$ but a complex step 3. Ko and Aluru came up with a scheme of classifying of each position in T as either "S" or "L", and picking the positions with one of these types (the one that appears the smallest number of times) for T'. This makes d at most $1/2$, usually smaller. Their idea was refined and simplified by Nong, Zhang, and Chan in two different ways. The variant they named *SA-IS* is the focus of this text.

3 S/L-types

We say that a position i of T is *S-type* if either $T[i...] < T[i+1...]$ (the suffix starting at i is *smaller* than suffix starting at the following position) or $i = n-1$ (the sentinel position). Otherwise (if the suffix starting at i is *larger* than suffix starting at the following position), we say that i is *L-type*. An *S-suffix* is a suffix that starts at an S-type position, and an *L-suffix* starts at an L-type position.

The following function computes the S/L-types of T in a boolean array *isS*, right-to-left in time $O(n)$:

```
const computeTypeArray = (T) => {
    const isS = Array(T.length);
    isS[T.length-1] = true;                      // sentinel
    for (let i = T.length-1; i > 0; i--) {
        isS[i-1] = T[i-1] < T[i] || T[i-1] === T[i] && isS[i];
```

```
    }
    return isS;
}
```

A key observation is that if two suffixes start with the same character, but one is S and the other is L, then the S-suffix is *larger* than the L-suffix. This is because if an S-suffix starts with character c, it must, after zero or more repetitions of c, be followed by a character that is *larger* than c. The opposite is true for an L-suffix: c must ultimately be followed by a *smaller* character. One (or both) of these larger/smaller characters determines the order of the two suffixes.

4 LMS-strings

We say that:

- A position i is an *LMS-position* (short for *leftmost S*) if $i > 0$, i is S-type, and $i-1$ is L-type. It is simple to determine if a position is an LMS-position from the type array. Note that there are at most $\lfloor n/2 \rfloor$ LMS-positions.
- An *LMS-suffix* is a suffix starting at an LMS-position.
- An *LMS-string* is a substring that starts with one LMS-position and ends with the next LMS-position, except for the sentinel which is an LMS-string on its own.

The reduction to a smaller problem (to solve recursively) is done by treating the LMS-strings as basic blocks, as if they were large characters, and creating a shorter string T' whose characters correspond to the LMS-strings.

Running example

T:	b	e	e	b	y	b	e	e	y	b	y	b	a	b	y	$
position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S/L:	S	L	L	S	L	S	S	S	L	S	L	L	S	S	L	S
LMS-positions:				↑		↑				↑			↑			↑
LMS-strings:				—	—	—				—	—	—	—			—
						—	—	—	—	—			—	—	—	—

LMS-string order

To be able use the sorted suffix order of T' for drawing conclusions about the suffix order of T, we need the order of the suffixes in T' to be the same as for the corresponding LMS-suffixes in T. For this purpose, we need to define the relative order of any two distinct LMS-strings to be the same as the order of the LMS-suffixes that start with them. Simply using lexicographic order *almost* does the trick, but one exception is needed: if one LMS-string is a proper prefix of the other, the longer one needs to be regarded as smaller than the shorter one.

For instance, consider the LMS-strings $T[3...5] =$ "byb" and $T[9...12] =$ "byba" in our example. $T[9...12]$ should be regarded as smaller than $T[3...5]$, because the suffix $T[9...] =$ "bybaby$" is smaller than $T[3...] =$ "bybeeybybaby$".

We get the order we need by comparing the two strings position by position, comparing characters first and S/L-types second. If characters match but S/L-types differ, we consider the string with L-type to be the smaller one, since an L-suffix is smaller than an S-suffix that begins with the same character.

5 Bucketing

I is an array of length n which ultimately (when the construction algorithm is finished) holds the suffix array. Its values are suffixes of T, represented by their starting positions $0, ..., n-1$. Saying that a suffix α is placed at position i of I is the same as saying that the integer $I[i]$ is set to the starting position of α. We also allow I to contain the value -1 for "none".

A *bucket* is a segment of I dedicated to suffixes with the same first character. There are as many buckets as there are different characters in T, and the size of the c-bucket is the same as the number of times c occurs in T.

Let the integer R be the current size of the alphabet (the maximum number of buckets). At the top recursion level, $R = |\Sigma|$, but when the algorithm invokes itself recursively, the alphabet is different from the original one, and then R can be both smaller or larger than $|\Sigma|$, although never larger than n.

We use an array B of length R to hold one position for each bucket. The following function, which is effectively the first part of *counting sort* (also known as *key-indexed counting*), sets B to either the first position (if the *end* parameter is false) or just after last position (if *end* is true) of each bucket.

```
const setBucket = (T, B, end) => {
    B.fill(0);
    for (const c of T) { B[c]++ }
    for (let i = 0, sum = 0; i < B.length; i++) {
        B[i] = (sum += B[i]) - (end ? 0 : B[i]);
    }
}
```

Since the length of B is R, both loops are executed at most $O(n)$ times under the assumption that $|\Sigma|$ is $O(n)$, and the running time of the function is $O(n)$.

6 Induced order

The key trick of the SA-IS algorithm is inducing the order of all suffixes from the order of only the LMS-suffixes in the following way. Say that we know the order of the LMS-suffixes. We place them in order in I, spread out at the ends of their buckets, and fill the rest of I with -1. In our example, the positions of the LMS-suffixes in order are:

```
15: $
12: aby$
 5: beeybybaby$
 9: bybaby$
 3: bybeeybybaby$
```

They are placed in I like this:

position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
bucket:	$	a	b						e				y			
I:	15	12	−1	−1	−1	5	9	3	−1	−1	−1	−1	−1	−1	−1	−1

Then we do:

1. Set bucket positions in B to the beginning of each bucket.
2. For each $I[i]$ from left to right: if $I[i]-1$ is an L-position, place it at the current position of its bucket, and advance the bucket position by one.
3. Set bucket positions in B to just past the end of each bucket.
4. For each $I[i]$ from right to left: if $I[i]-1$ is an S-position, decrease its bucket position by one, and place $I[i]-1$ in that position.

In Javascript:

```
const induceOrder = (isS, I, T, B) => {
    setBucket(T, B, false);
    for (let i = 0; i < T.length; i++) {
        const j = I[i]-1;
        if (isS[j] === false) { I[B[T[j]]++] = j }
    }
    setBucket(T, B, true);
    for (let i = T.length-1; i >= 0; i--) {
        const j = I[i]-1;
        if (isS[j]) { I[--B[T[j]]] = j }
    }
}
```

Let us look at what effect step 2 has on our example. We start the scan with $I[0]$ which is 15, and find that $15-1=14$ is an L-position. Suffix $T[14...]$ should go in the "y"-bucket (because $T[14]$ is "y") which starts at position 12, so we place 14 at $I[12]$ and advance the "y"-bucket to 13. Then we look at $I[1]$ which is 12, and $12-1=11$ is also an L-position, so we place 11 at $I[2]$, the beginning of the "b"-bucket (because $T[11]$ is "b"), and advance the "b"-bucket to 3. Then we look at $I[2]$, which we just set to 11, and again $11-1=10$ is an L-position, and since $T[10]$ is "y", we place 10 at $I[13]$, the next position of the "y"-bucket, and advance the "y"-bucket to 14. Then we look at $I[3]$, which is -1, and since $-1-1=-2$ is not an L-position, we place nothing. And so on, until we get to the end of I, which then contains:

position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
bucket:	$	a	b						e				y			
I:	15	12	11	−1	−1	5	9	3	2	1	−1	−1	14	10	4	8

Note that in addition to the LMS-positions, I now contains all the L-positions, in the correct suffix order. Also, they are grouped together at the beginning of their buckets, and since L-suffixes are smaller than S-suffixes that begin with the same character, they are actually in their correct final positions in the suffix array.

This may seem remarkable, but it is no coincidence. We now prove that this is always the case at the end of step 2.

Proof of L-inducing correctness

Consider what happens in an iteration of step 2 that encounters suffix α in $I[i]$. Assume that an L-suffix starts one position to the left of α (otherwise nothing changes in the current iteration), and call that L-suffix $\beta = c\alpha$. We then place β at the next free position of the c-bucket, which is the correct position for β if any $c\alpha' < c\alpha$ have already been placed and no $c\alpha'' > c\alpha$ have been placed. This is guaranteed if α', α, and α'' are encountered in order according to their correct positions, or in other words, if the suffix that has correct position i is always placed there before the scan reaches i.

This is obviously true for $i = 0$. We prove that if it is true for $i - 1$, it must alse be true for i, and therefore (by induction) it is true for *any* i.

Assume that any suffix whose correct position is smaller than i has been placed when the scan reaches i. Let β be a suffix whose correct position is i. If β is an LMS-suffix, it was already correctly placed when the scan began, so we only need to consider the case that $\beta = c\alpha$ is an L-suffix. Then α is either

- an L-suffix, which is smaller than β, because β is an L-suffix; or
- an S-suffix whose first character is smaller than c. (It cannot be equal to c, because then β would also be an S-suffix.) Then α is an LMS-suffix (because it follows an L-position), whose position in I is before the c-bucket.

In either case, α has correct position $j < i$, and β was put in place when the scan encountered α in $I[j]$.

S-inducing

Step 4 goes in the opposite direction of step 2, and results in all S-suffixes being put in place. We omit the correctness proof, which follows the same reasoning as the correctness proof for the of L-inducing correctness. (Making this explicit is a good exercise!)

Time

The induce procedure consists of two bucket initializations and two loops through I. Each runs in $O(n)$ time, which makes the total time also $O(n)$.

7 LMS-string sorting

To go from T to the reduced string T', we pick out the LMS-positions of T, and translate the LMS-string in each of those positions to a character in a new alphabet. The order of the characters in the new alphabet needs to respect the order of the correpsonding LMS-strings. To find the right alphabet, we sort the LMS-strings, and then assign them numbers in increasing order.

A major simplification in the SA-IS algorithm compared to the algorithm of Ko and Aluru comes from the observation that the same induce procedure that we have already covered can be used for sorting the LMS-strings. We spread out the LMS-positions over the ends of their buckets, and fill the rest of I with -1. (This is just like when setting up for inducing suffix order, except that now

the LMS-strings are not sorted apart for the bucketing on the first characters). Then we run the induce procedure. At the end of step 2, L-suffixes are in place, sorted on their prefixes up to and including the first S-position. At the end of step 4, the S-suffixes have also been put in place, and the order between those of them that start at LMS-positions are in our desired LMS-string order.

We omit the proof, but again it follows the same pattern of reasoning as the correctness proof for inducing suffix order.

8 Suffix sorting a string with all distinct characters

Recursive computation of suffix arrays can stop when all characters of T are distinct, because then suffix sorting only needs to consider the first character of each suffix, and we can find it simply as the inverse permutation of T in $O(n)$ time like this:

```
const inversePermutation = (T) => {
    const I = Array(T.length);
    for (let i = 0; i < T.length; i++) { I[T[i]] = i }
    return I;
}
```

9 Putting it together

Now we are ready for the main suffix sorting function, which approximately follows the sample implementation of Nong, Zhang, and Chan. It takes two arguments: an array T of length n containing the string, and an integer R such that $T[n-1] = 0$ and $1 \leq T[i] < R$ for $i = 1, ..., n-2$.

```
export const computeSuffixArray = (T, R) => {
```

First compute the type array and define a function to check whether a given i is an LMS-position. Allocate I and the bucket array B.

```
    const isS = computeTypeArray(T);

    const isLMS = (i) => isS[i] && isS[i-1] === false;

    const I = Array(T.length);                  // position array
    const B = Array(R);                         // bucket array
```

Spread out the LMS-positions in I, on the right sides of their buckets, with the rest of I filled with –1. Then use the induce procedure to obtain the order of LMS-strings.

```
    setBucket(T, B, true);                      // find ends of buckets
    I.fill(-1);                                 // -1 is empty position
    let n  = 0;                                // length of T
    for (let i = 1; i < T.length; i++) {
```

```
        if (isLMS(i)) {
            I[--B[T[i]]] = i;
            n ++;
        }
    }

    induceOrder(isS, I, T, B);                  // induce order of LMS-
strings
```

Now the LMS-positions are in I in LMS-string order. Allocate T' and use it temporarily to hold the LMS-positions in correct order.

```
    const T  = Array(n );
    for (let i = 0, j = 0; i < T.length; i++) {
        if (isLMS(I[i])) { T [j++] = I[i] }
    }
```

Use I temporarily to map positions in T to LMS-string numbers, assigned in increasing order. We scan T' to find the LMS-positions. For each position, we check whether it points to an LMS-string that is different from its predecessor, in which case we increase the counter for distinct LMS-strings. We then place the character number of the current LMS-string (one less than the counter) in the position of the LMS-string in I. This will let us use I for mapping the LMS-position to the corresponding character in T'.

```
    I.fill(-1);                                 // set empty again
    I[T [0]] = 0;                               // assign 0 to sentinel
    let R  = 1;                                 // counts assigned names
    for (let i = 1, p = T [0]; i < n ; i++) {
        const q = T [i];                        // the LMS-position to consider

        // Compare q to previous string p, by character and (if equal) S/L.
        for (let d = 0;; d++) {
            if (T[p+d] !== T[q+d] || isS[p+d] !== isS[q+d]) {
                R ++;                           // p different from q
                p = q;                          // compare to this one instead
                break;                          // done with q
            }
            if (d > 0 && isLMS(p+d)) { break }  // LMS-string ends? done with q
        }
        I[q] = R -1;                            // map position to T  character
    }
```

Finalize T' by getting the LMS-string characters from I.

```
    for (let i = 0, j = 0; j < n ; i++) {
        if (I[i] >= 0) { T [j++] = I[i] }
    }
```

Compute I', the suffix array of T'. If the length of T' is equal to the number of distinct LMS-strings, all characters in T' are distinct, and we can find I' as the inverse permutation of T'. Otherwise, we invoke the suffix array computation recursively for T'.

```
    const I  = R  < n  ? computeSuffixArray(T , R ) : inversePermutation(T );
```

Reuse T' again for holding the LMS-positions of T. Then use T' as a map to translate the contents of I' from T' positions to the corresponding T positions.

```
    for (let i = 0, j = 0; j < n ; i++) {
        if (isLMS(i)) { T [j++] = i }
    }
    for (let i = 0; i < n ; i++) { I [i] = T [I [i]] }
```

Spread out the sorted LMS-suffixes at the ends of buckets in I. Then use the induce procedure to obtain the order of all suffixes, and return the result.

```
    setBucket(T, B, true);
    I.fill(-1);
    for (let i = n -1; i >= 0; i--) {
        const j = I [i];
        I[--B[T[j]]] = j;
    }

    induceOrder(isS, I, T, B);

    return I;
}
```

Analysis

Apart from the recursive call, each stage of the suffix array computation consists of a loop with at most n constant-time steps, requiring in total $kn + k'$ time for some constants k and k'. The recursive call performs suffix array computation on a string of length at most $\lfloor n/2 \rfloor$. In total, therefore, the time complexity as a function of n is bounded by

$$\begin{aligned}
time(n) &\leq kn + k' + time(\lfloor n/2 \rfloor) \\
&\leq kn + k' + \frac{kn}{2} + k' + time(\lfloor n/4 \rfloor) \\
&\leq kn + k' + \frac{kn}{2} + k' + \frac{kn}{4} + k' + time(\lfloor n/8 \rfloor) \\
&\leq \sum_{i=0}^{\lceil \lg n \rceil} kn \cdot 2^{-i} + k' \\
&\leq 2 \cdot 2kn + (\lceil \lg n \rceil + 1)k'
\end{aligned}$$

The running time is therefore $O(n)$. Allocated storage space follows the exact same analysis (only a constant number of arrays of size $O(n)$ are allocated), and is therefore also $\mathrm{O}(n)$.

10 Using less space

It may be desirable to reduce the $O(n)$ extra space used in the algorithm (for the S/L-type array, the bucket array, and for T' and I'). After all, a suffix *tree*

takes $O(n)$ space too, and the main reason for preferring a suffix array is that it requires less space.

Smaller constant

The algorithm as stated in above already reuses the I and T' arrays for a couple of different purposes instead of creating a new array all the time. The example C code ofof Nong, Zhang, and Chan goes further in reducing space:

- The bottom and top parts of I are reused to contain I' and T' during the recursive call. In order to simultaneously hold both the map from T' characters to LMS-positions and the LMS-positions themselves in the reduction stage, it uses the trick of dividing the LMS-positions by two (which works because LMS-positions are never right next to each other), reducing the space for the map to half.
- The bucket array is deallocated before the recursive call, and then allocated again after the suffix array for T' is created. Thereby, only one bucket array exists at a time.
- The S/L-type array is packed into the individual bits of a byte array. Although only one bit is necessary to hold a Boolean, unfortunately almost all programming environments require *at least* a byte per array element, which means that this is reduction by a factor eight or more for the S/L-type arrays.

Less than linear auxiliary space

With some more elaborate tricks, it is possible to do away with the n-length auxiliary arrays completely.

- Nong, one of the authors behind the SA-IS algorithm, presented a variant that requires only $O(|\Sigma|)$ auxiliary space, which is needed for the bucket array at top recursion level.
- Li, Li, and Huo presented variants that use only $O(1)$ auxiliary space for integer alphabets, even when T is read only.