Lecture 21: Suffix Array Construction with SA-IS (Updated 3)

Sam McCauley

December 7, 2021

1 Some String Preliminaries

A string is a sequence of characters. Each character of a string must be from some *alphabet.* Oftentimes, especially in `C`, this alphabet is the set of all 256 ASCII characters (though we'll need to consider more general alphabets today).

We'll use array indexing to refer to specific characters in a string, and assume that they are 0-indexed. So if string S stores `hello`, we have $S[0] =$ `h` and $S[1] =$ `e`.

A *substring* of a string S is a sequence of consecutive characters of S. If a substring starts at character 0 it is a *prefix* of S; if it ends at the last character it is a *suffix* of S. We call the suffix starting at character i the *ith suffix* of S. (So, for example, the 0th suffix of S is S itself; the 1st suffix consists of S with its first character removed.)

We assume that the character `$` does not appear in the alphabet; furthermore we assume that all strings end with `$`. (Of course, this assumption is easy to maintain by appending `$` (or some other non-alphabet character) to any string that does not satisfy the assumption.) We consider `$` to be before any other character in sorted (i.e. lexicographic) order. On your assignment you will be given strings that end with character `'0'`, so this assumption will be already satisfied for all strings you are given in your code.

2 Suffix Arrays

One of the most ubiquitous string problems involves searching for occurrences of one string in another. In particular, let's say you have a large text T. You want to preprocess T (using at most $O(|T|)$ space) so that for any pattern P, you can quickly determine if P is a substring of T.

Here's one way we could solve this problem. Let's sort all the *suffixes* of T and store them in an array. So if T was `banana`, our suffixes would be, in sorted order: `a`, `ana`, `anana`, `banana`, `na`, `nana`. The bad news is that this requires $O(n^2)$ space—but the good news is that we can use binary search to answer queries very quickly!

Let's say P occurs at character i of T—so $T[i] = P[0]$, $T[i+1] = P[1]$, and so on until $T[i+m-1] = P[m-1]$. But this means that the first m characters of suffix i of T match P. That is to say: P occurs in T if and only if P is a prefix of some suffix of T.

Therefore, we want to find some suffix of T that starts with P (i.e. has P has a prefix). Since we have a sorted table of these suffixes handy, we can find such a suffix using binary search. Let's say that $|T| = n$ and $|P| = m$. Each iteration of the binary search involves a comparison between P and a suffix of T: this requires $O(m)$ time in the worst case. We are binary searching over n suffixes, so we require $O(\log n)$ iterations for the binary search to complete. Overall, our table allows us to determine if P is a substring of T in $O(m \log n)$ time—not bad!

```
b a n a n a $
  a n a n a $
    n a n a $
      a n a $
        n a $
          a $
          a $
            $
```

Figure 1: Let's say P is `ana` and T is `banana$`. P does appear in T since it is the prefix of a suffix of T—in particular, `ana` is a prefix of `anana$`. In fact, P appears twice in `banana`, and indeed `ana` is a prefix of another suffix of `banana`: `ana$`.

Improving this result. The glaring issue with the above is space usage. The key observation here is that storing all suffixes explicitly is extremely wasteful since we have access to T. In fact, I can just store i (the index of a given suffix), and the suffix can be easily obtained character-by-character by looking in T.

So let's store the *indices* of the suffixes in sorted order. So for `banana$`, rather than storing `$`, `a$`, `ana$`, `anana$`, `banana$`, `na$`, `nana$`, we would store their indices: 6, 5, 3, 1, 0, 4, 2. This data structure is called a *suffix array.*

Note that our suffix array allows our binary search to continue as planned. Let's call our suffix array SA. We want to do a binary search in SA. When comparing P to the ith entry of SA, we use T to recover the characters: we compare $P[0]$ to $T[SA[i]]$; if they are equal we compare $P[1]$ to $T[SA[i]+1]$, and so on. A single comparison requires $O(m)$ time, and we perform $O(\log n)$ comparisons, for $O(m \log n)$ total time. This performance is achieved with an $O(n)$ space suffix array.

While we won't discuss it here, it's possible to extend the suffix array to answer these queries in $O(m)$ time—the time to find an occurrence of P in T is asymptotically the same as the time just to *read* P!

Goal for Today. How fast can we build a suffix array? Sorting the suffixes requires $O(n^2 \log n)$ time (we are sorting n suffixes, and comparing any two suffixes may requre $O(n)$ time). Of course, there's a lot of redundancy in those comparisons—certainly it seems plausible that we can compare suffixes in $O(1)$ time on average, achieving $O(n \log n)$ time.

Somewhat shockingly, we can in fact do much better. The suffix array can be built in $O(n)$ time so long as the size of the alphabet is no larger than n.[1]

Side note: It's worth mentioning that we can, alternatively, build a *trie* on the suffixes of T, as in Figure 2. (You may not have seen a trie before—don't worry if not; we won't be going over it beyond this side note.) A cleverly-built trie will also require $O(n)$ total space, and enable us to determine if P occurrs in T (with a walk through the tree) in $O(m)$ time. This data structure is also quite popular—in short, it's a bit easier to work with than a suffix array, but maintaining the tree structure leads to significantly larger constants in space consumption. In any case, an easy, efficient algorithm allows a suffix tree to be transformed into a suffix array and vice versa in $O(n)$ time—so the lessons learned here can be applied to suffix trees as well.

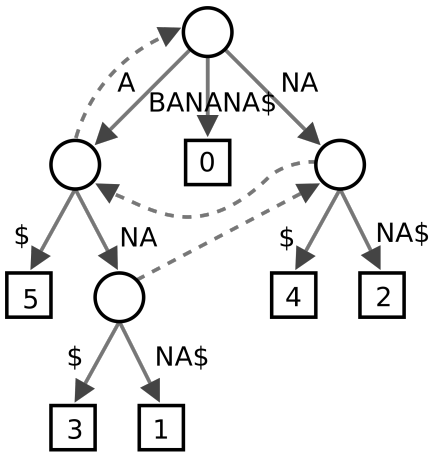


Figure 2: The suffix tree (pictured here) is a space-inefficient alternative to the suffix array. We'll be focusing on suffix arrays, but most of the basic principles apply to both.

Generalizations. In fact, the suffix array can solve far more general problems. You may notice that with two binary searches we can count the *number* of occurrences of P in T (still with $O(m \log n)$ total time). We can also list all locations of P in T in essentially the same running time. The suffix array can be easily generalized to handle the case where we have several texts $T_1, T_2, \ldots$ and we want to find occurrences of P in any of these texts.

Of course, we recently saw another application: the suffix array allows us to calculate the Burrows-Wheeler transform of a string in $O(n)$ time, leading to effective compression.

In fact, the suffix array allows us to answer some quite surprising questions very quickly. We can use a suffix array to find the longest common substring between two strings S_1 and S_2 in $O(S_1 + S_2)$ time. We can find the longest palindrome in a string S in $O(|S|)$ time. (Note that just *verifying* that a given substring is a palindrome may take as long as $O(|S|)$ time.)

The suffix array can be combined with other data structures to answer even more complicated queries. For example, the suffix array can be used to search for occurrences of P in T *with errors*: is there a substring

[1] In fact, the algorithm can easily handle significantly larger alphabets—but alphabets of size $\omega(n)$ require an extra preprocessing step so we'll ignore them here.

of T that matches P in all but k characters?

2.1 History of Suffix Array Construction Algorithms

The suffix tree was first invented by McCreight in 1976, building on earlier ideas of Weiner. The construction algorithm was improved by Ukkonen in 1995. All of these algorithms constructed the suffix tree in $O(n)$ time for constant-sized alphabets using *suffix links*—carefully chosen pointers in the data structure to help jump to closely-related suffixes in other parts of the tree. (These are drawn using dotted lines in Figure 2.)

As mentioned above, suffix trees are space-inefficient (an issue that is exacerbated by use cases that require suffix links). This motivated Manber and Myers to create the *suffix array* in 1990, pointing out in the abstract of their paper that the suffix array is "3 to 5 times more space-efficient" than a suffix tree. They also gave an $O(n)$ time algorithm for construction for constant-sized alphabets.

Achieving $O(n)$ time for non-constant sized alphabets remained open. Note that the motivation behind attaining this bound was twofold. First, of course, it was important for strings with very large alphabets. But on top of that, even for moderately-sized alphabets (like one would see in human-generated text), extra terms that depend on the alphabet slow down the running time fairly considerably. As we'll see, generating these data structures is fairly computationally intensive—an extra factor of (say) 5 or 10 in the running time is quite noticeable in practice.

Linear-time Algorithms Through Recursion The breakthrough came in 1997 with the suffix tree construction algorithm of Farch-Colton, that ran in $O(n)$ time for any alphabet of size at most n. Unlike previous methods, this idea did not use extensive bookkeeping in the form of suffix links. Instead, it approached the problem *recursively.*

Here's the idea of how Farach-Colton's algorithm works on a string S of size n.[2] Let's look at only the *odd-numbered* suffixes of S: the suffixes starting at indices 1, 3, 5, and so on.

Here's the idea of the algorithm. Let's sort the odd-numbered suffixes recursively.[3] (The base case is when there's just one odd-numbered suffix, which is trivial to sort.)

We can assume (recursively) that the odd-numbered suffixes are sorted. Now, we want to finish building the suffix array. This is where the really clever part of the algorithm comes in. I won't go into the details of this initial result here, but we ARE going to see a recursive method for building the suffix array today, so you'll see one example of how this can work.

Some thoughts on the combining phase for now: how can an ordering of the odd-numbered suffixes help us order the even-numbered suffixes? On the one hand, this seems impossible: the even-numbered suffixes all start with entirely different characters than the odd-numbered suffixes, so the ordering is, right off the bat, 100% different. On the other hand, this first character is the *only* difference: once we sort the even-numbered suffixes by their direct character, the odd-numbered suffixes determine the rest of their ordering. Ultimately, it turns out that (with some care and bookkeeping), the odd-numbered suffixes can be used to sort the even-numbered suffixes in $O(n)$ time.

How long does this algorithm take? It's a recursive algorithm, so let's write out the recurrence; let's say $T(n)$ is the time this recursive algorithm requires on a string of length n. The number of odd-numbered suffixes is at most $n/2$; the work done to achieve the final result after the recursive call is $O(n)$. So we have $T(n) = T(n/2) + O(n)$ (with base case $T(n) = O(1)$ if $n \leq 2$). This solves to $T(n) = O(n)$.

Karkkainen and Sanders gave a simpler version of the recursive approach with slightly worse runnign time. In short, they split the string's suffixes into three groups rather than two, and recurse on two of the three groups. This means that the remaining suffixes are "sandwiched" between two sorted suffixes, simplifying the combine step. The cost is that the recursive calls are significantly larger, leading to somewhat slower running time. An excellent exposition of this algorithm can be found in Erik Demaine's advanced algorithms class at MIT; a video lecture and scribe notes can be found here: `http://courses.csail.mit.edu/6.851/spring12/lectures/L16.html`.

[2] We'll discuss it as if it were a suffix array algorithm, though it was originally presented for suffix trees.

[3] You may notice that I'm sweeping some details under the rug: this isn't *quite* recursive, since our original goal was to sort all suffixes of a string S; the odd-numbered suffixes are not the collection of suffixes of a single string.

The State of the Art The recursive approach revolutionized suffix array algorithms, and the main line of research since then has focused on improving these algorithms. Generally, these improvements took the form of better recursive performance at the cost of extra bookkeeping.

Today, we'll be learning about the **SA-IS algorithm** of Nong, Zhang, and Chan. This algorithm:

- Uses only arrays to calculate the suffix array—in fact, it only needs the suffix array itself along with a couple of temporary arrays;
- Decreases the input size by 2 with each recursive call in the *worst case*—in practice the size often decreases by 3-4;
- Is quite simple (though precise)—it depends on a few straightforward subroutines and can be written in a couple hundred lines of code;
- Has (in short) the best practical performance of any algorithm.

There has been further research since then, modifying SA-IS to improve running time at a few tricky parts, or improving the space efficiency, or extending the ideas to other data structures. Today we'll be focusing on SA-IS how it was originally presented.

3 The SA-IS Algorithm

While it's simple in terms of avoiding too much bookkeeping, the SA-IS (suffix array induced sort) algorithm has a number of different parts—parts that may not make sense until they're all put together at the end. So this explanation will be a bit out of order. Here's the order we'll use to discuss the algorithm:[4]

First, we'll talk about some basic principles and vocabulary that we'll be using in the algorithm (Section 3.1).

Then, we'll skip the recursive call and go right to obtaining the final solution—we'll talk about how if we obtain the sorted order of special subset of the suffixes (the LMS suffixes), we can finish finding the suffix array in $O(n)$ time. This step will be called an *induced sort* (Section 4.1).

Finally, we'll talk about the recursive call: how can we sort the LMS suffixes (Section 5)? This is split into further parts. First, we'll split the LMS suffixes into substrings (the LMS blocks). We'll sort the LMS blocks, and use the result to relabel the LMS suffixes with a new alphabet. Finally, we'll use this relabelling to construct a new string, and recursively find the suffix array on this string; this suffix array will give us the order of the LMS suffixes.

That's a lot of explanation for a (supposedly simple) 100 lines of code! In short: the SA-IS algorithm is fairly simple in terms of methodology, but we'll be going over *how* it works.

In all of the below, we'll assume that we want to find the suffix array of a string S of length n, storing the result in a suffix array SA.

	b	b	b	b	b	c
i						
	b	b	b	b	c	
$i+1$						

Figure 3: Comparing suffix i to suffix $i+1$ (only a part of each suffix is shown). In this case, $S[i] = S[i+1]$. Because the next character not equal to `b` is `c`, suffix i is S-type. If there had been an `a` instead of a `c`, suffix i would have been L-type.

3.1 Basic Principles: L-type, S-type, and LMS suffixes

A suffix $i < n-1$ is S-type if it is *smaller* in lexicographic order than suffix $i+1$; it is L-type if it is *larger* in lexicographic order than suffix $i+1$. The last suffix is always considered S-type.

The first step of our algorithm is going to be labelling every suffix as S-type or L-type. How can we do that in linear time? Let's look at two different cases.

[4]This presentation of SA-IS was adapted from Keith Schwarz' excellent slides here: `http://web.stanford.edu/class/archive/cs/cs166/cs166.1196/lectures/04/Small04.pdf`.

- First, let's say the second character of the ith suffix is different from the first; i.e. $S[i] \neq S[i+1]$. If $S[i] < S[i+1]$ then the ith suffix is S-type; if $S[i] > S[i+1]$ then the ith suffix is L-type.
- If the second character of the ith suffix is equal to the first, then the type is determined by the next character not equal to $S[i]$. Here's one way we can simplify this: if $S[i] = S[i+1]$, then the type of the ith suffix is equal to the type of the $i+1$st suffix.

Putting these together, we obtain a linear-time algorithm. We scan S from back to front. The last character is always S type. Then, for each character $S[i]$, we compare $S[i]$ and $S[i+1]$. If $S[i] < S[i+1]$ we store that i is S-type; if $S[i] > S[i+1]$ we store that i is L-type; if $S[i] = S[i+1]$ we copy the type of suffix $i+1$ over to i.

With these types in mind, we define LMS suffixes (the basic building block of the SA-IS algorithm). A suffix i is an LMS suffix if i is the last character in S, or if suffix i is of type S and suffix $i-1$ is of type L. (Suffix 0 can never be an LMS suffix.) These can easily be identified in linear time.

Let's go ahead and label the L and S suffixes of an example string: `CGACTCCAACAACAAGCT$`.

	S	L	S	S	L	L	L	S	S	L	S	S	L	S	S	L	S	L	S
	C	G	A	C	T	C	C	A	A	C	A	A	C	A	A	G	C	T	$
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Figure 4: Labelling a string with the type of each suffix (L or S). The LMS suffixes are highlighted in red.

What can we say, already, about the final suffix tree of our string? Right off the bat we may observe that the sorted suffixes must be grouped by their first letter:

	$	A						C								G		T	
SA:	18	7	10	13	8	11	2	14	6	9	12	5	0	16	3	1	15	17	4

String S:	S	L	S	S	L	L	L	S	S	L	S	S	L	S	S	L	S	L	S
	C	G	A	C	T	C	C	A	A	C	A	A	C	A	A	G	C	T	$
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Figure 5: Looking ahead to the final suffix array, we can see that suffixes are grouped by their first character.

Furthermore, let's look at how the L-type and S-type suffixes wind up being placed in the final suffix array. In Figure 6 we can see that the L suffixes are always placed at the front of the bucket, whereas the S suffixes are always placed in the end of the bucket.

We can formalize this idea with the following lemma.

Lemma 1. *Consider two suffixes i and j of S such that i is L-type, j is S-type, and both i and j start with the same character ($S[i] = S[j]$). Then the ith suffix is before the jth suffix in sorted order.*

With the above, in $O(n)$ time, we can:

- Count the characters to determine where each "bucket" of suffixes that start with the same character begins and ends,
- label each suffix as L-type or S-type and find the LMS suffixes,
- place each suffix in the correct character bucket with the other L-type or S-type suffixes.

Therefore, all we have left to do is sort the suffixes that are of the same type and start with the same letter.

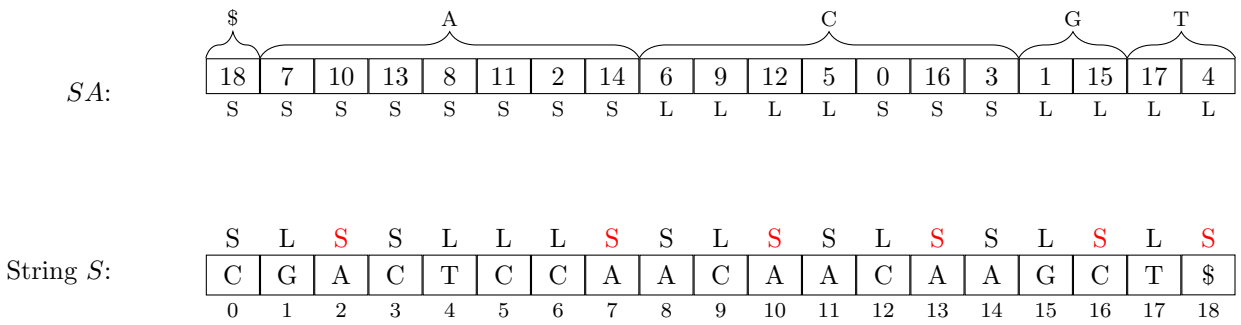


Figure 6: Looking ahead to the final suffix array, the L-type suffixes are always before all S-type suffixes in their bucket.

4 Building the Final Suffix Array Using Induced Sorting

In this section, we skip ahead in terms of the final algorithm. **Let's assume that the LMS suffixes are stored in sorted order.** In Section 5 we will describe how the LMS suffixes can be sorted—but for now let's just leave it as an assumption.

Let P be an array containing the indices of the LMS suffixes so that the suffixes are stored in lexicographical order. The goal of this section is to use P to determine the sorted order of *all* suffixes in $O(n)$ time.

Let's start small: we have P in sorted order. Can we say anything about the order of the other suffixes of S?

Well, yes, we can say a little bit. Each LMS suffix i is preceded by an L-type suffix $i-1$. By definition of L-type suffix, suffix $i-1$ is after suffix i in sorted order. In fact we can extend this even further: if suffix $i-2$ is L-type, it must be even larger, and so on. By definition, any sequence of L-type suffixes is followed by an LMS suffix. This means that any L-type suffix can be related to an LMS suffix, as in Figure 7. For example, we know that since suffix 6 is L-type, suffix 6 is larger than suffix 7; furthermore, since suffix 5 is L-type, suffix 5 is larger than suffix 6.

OK, so now we've drawn a big graph of suffixes. But, for the most part, we don't know very much about the sorted order at all. For example, we know nothing about how suffix 12 compares to suffix 5. (In fact, all we know is that suffix 12 comes before suffix 13, and suffix 5 comes before suffix 7—but suffix 12 and 5 both start with C, whereas suffixes 13 and 7 start with A.) How can we get the sorted order in $O(n)$ time?

One way to start is by noticing that Figure 7 is essentially an (unbalanced) heap. We can see immediately that 18 must be the smallest suffix in the whole string. After that, either 7 or 17 is the smallest remaining suffix. Comparing, we see that 7 is smaller. Then, 17 or 6 or 10 must the smallest suffix. But we have a problem here: the number of suffixes we need to compare each time is growing. And even comparing *two* of these suffixes may require $O(n)$ time—but we only have that much time to do the entire sort!

Instead, we use the first character of each suffix to determine where it will go. In fact, we'll find that in all cases, we can place each new suffix in our "heap" above without doing any suffix comparisons at all! Repeatedly removing the smallest suffix will allow us to sort all L-type suffixes.

Example sort. Let's actually sort the suffixes from Figure 7, building up the algorithm as we go. The final induced sort won't look quite like this—this is just to motivate our ultimate approach. For the sake of space, let's only include some intermediate states (rather than redrawing the diagram each time a suffix is moved).

As mentioned, in Figure 7, we can immediately see that suffix 18 is the smallest. Now, where does suffix 17 go? By looking at $S[17]$, we can see that suffix 17 starts with a T. That's easy: suffix 17 goes at the very bottom of the rightmost column; it comes later than suffix 16.

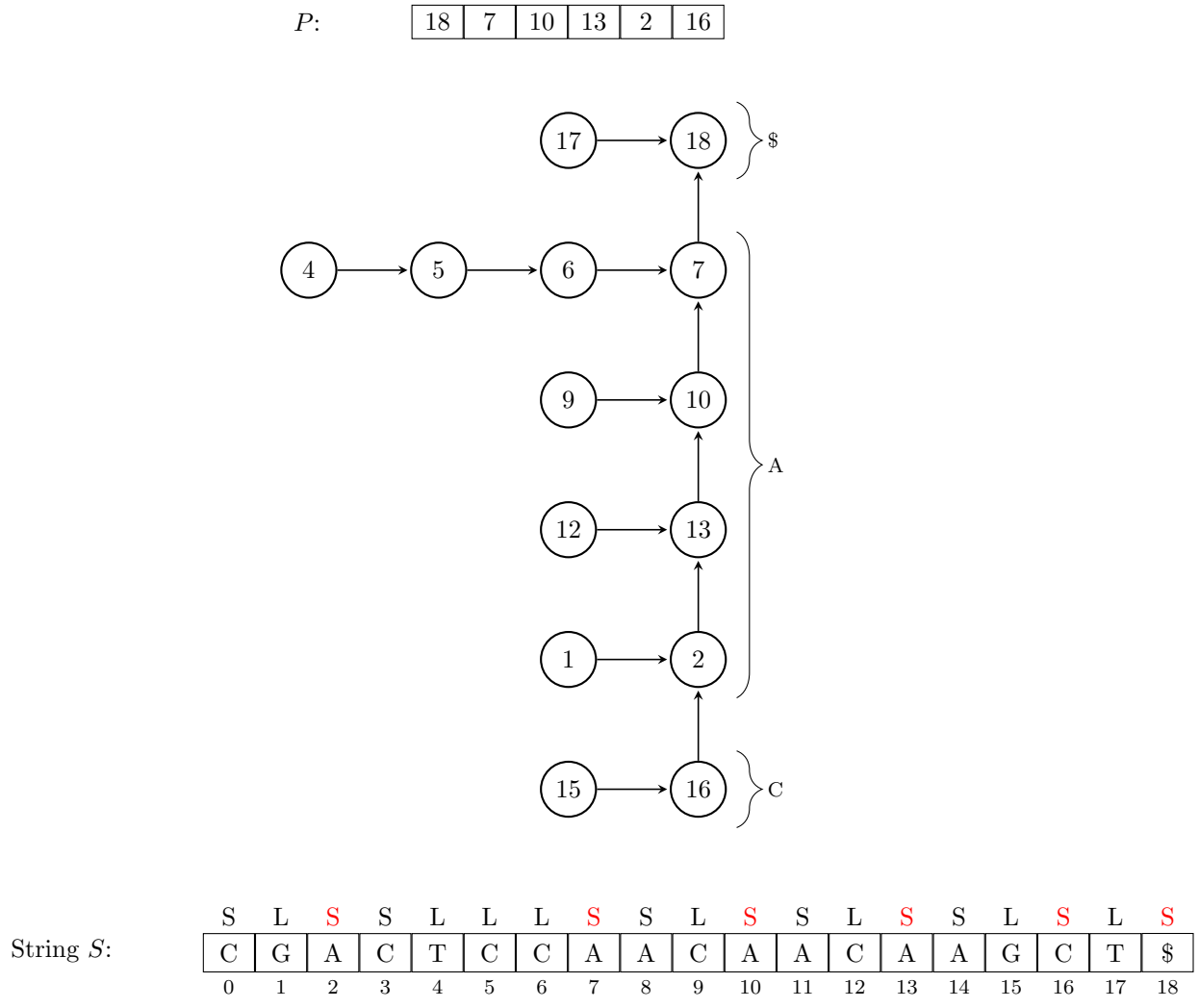


Figure 7: We know each L-type suffix is larger than the suffix that comes after it. We can represent all of these dependencies graphically as above. (There is an arrow from suffix i to suffix j if i comes after j in sorted order.) The rightmost column of nodes consists of the LMS suffixes.

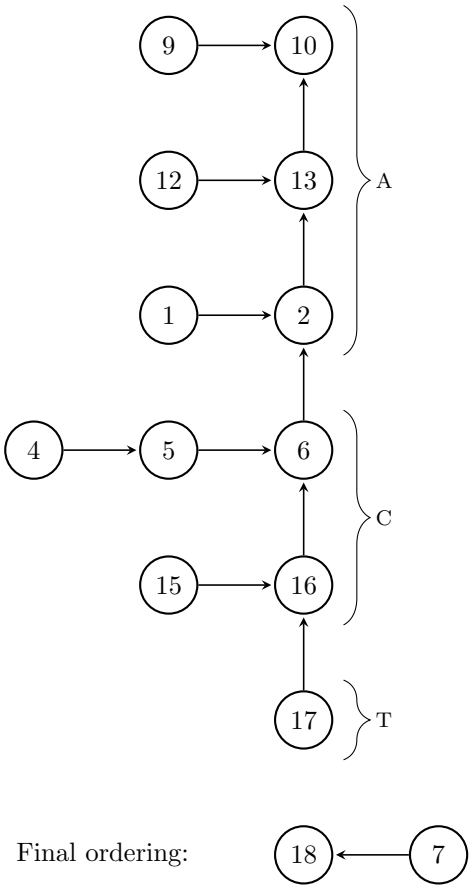
Since suffix 18 has been placed, and suffix 17 is now lower in our "heap", we know that 7 is the next smallest. Where do we put suffix 6? Looking at $S[6]$, we see that suffix 6 starts with C. But, how does it compare to suffix 16? Since suffix 16 is an LMS suffix, it is S-type. Suffix 6, however, is L-type. Therefore, we know by Lemma 1 that suffix 6 comes before suffix 16. Let's move suffix 6 where it belongs (bringing suffixes 4 and 5 along with it).

Let's draw where we are right now in Figure 8.

Suffix 10 can, then, be written to the final ordering directly. Now we need to place suffix 9. Suffix 9 starts with C and is L-type. Let's look at where we are again in Figure 9 (on page 9).

We know that suffix 9 is before suffix 16 (because 9 is L-type and 16 is S-type). How does suffix 9 compare to suffix 6?

To answer this question, we need to think a little bit about what these suffixes really are. Suffix 9 consists of a C followed by suffix 10. Suffix 6 consists of a C followed by suffix 7. So the ordering of suffix 9 and suffix 6 is the same as the ordering of suffix 10 and suffix 7. But we already know this ordering from the



String S:

	S	L	S	S	L	L	L	S	S	L	S	S	L	S	S	L	S	L	S
	C	G	A	C	T	C	C	A	A	C	A	A	C	A	A	G	C	T	$
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Figure 8: We were able to correctly order suffixes 18 and 7. Suffix 6 was placed with the other C suffixes later in the tree; since it was L type we placed it before all S-type suffixes that start with C.

final ordering—suffix 7 comes first! Therefore, suffix 9 goes behind suffix 6.

In fact, we can generalize to the following observation: each time we want to place a new L-type suffix i with first character c, i goes in front of all S-type suffixes with first character c, but goes after all L-type suffixes with first character C.

Notice that in the above example, we have done *no string comparisons whatsoever*. Each iteration, we place one suffix in the correct final ordering. Any suffixes that are linked to it from the left are placed in the correct position in the rightmost column—this can be done by looking only at the character and the type of this suffix.

Iterating further, we obtain a final order (for all L-type suffixes and all LMS suffixes) of:

18	7	10	13	2	6	9	12	5	16	1	15	17	4

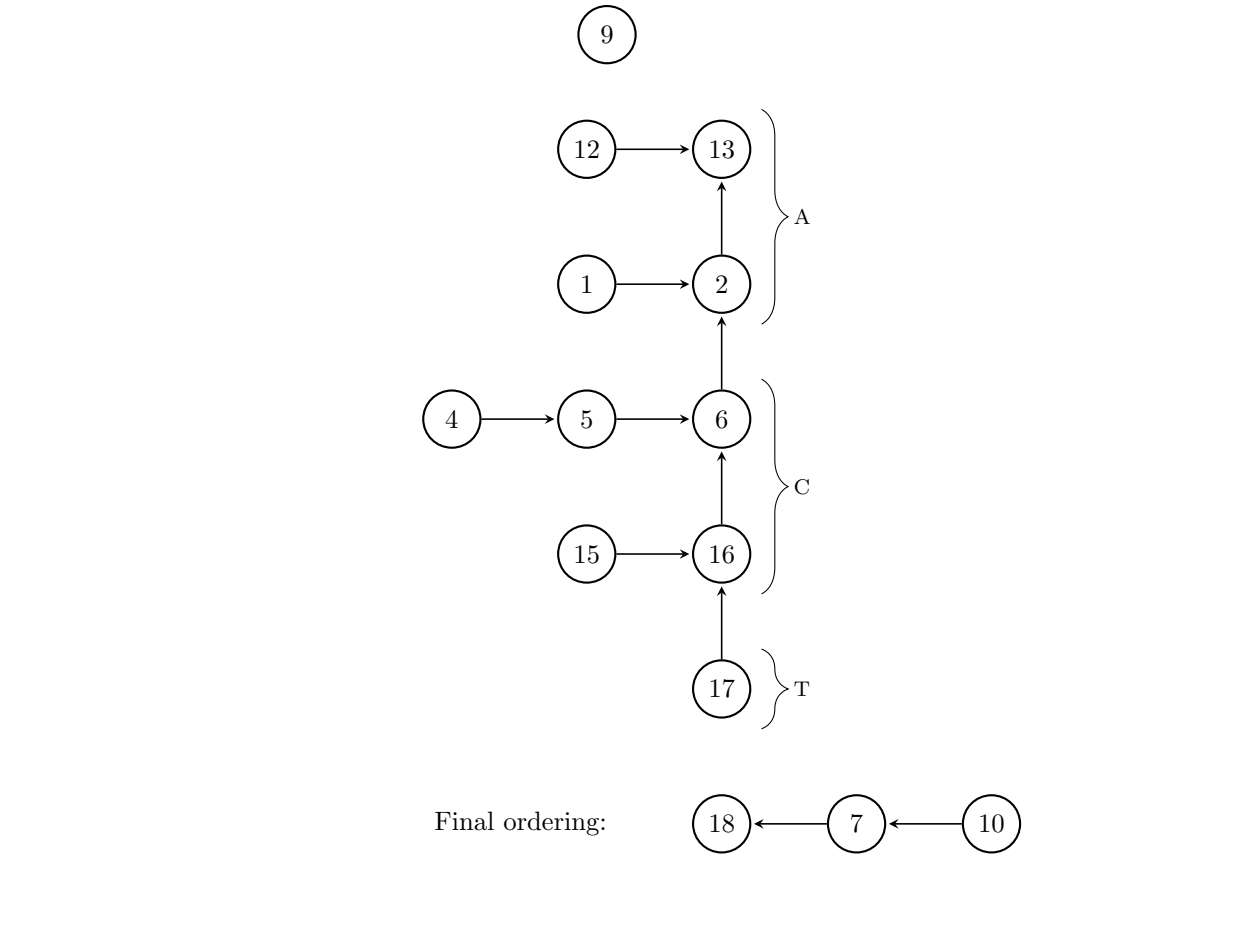


Figure 9: We were able to use the first character and type of each suffix to reorder all the suffixes until 9, which we now need to place. Suffix 9 starts with C and is of L-type—so we know it comes after suffix 2 (which starts with A) and before suffix 16 (which starts with C and is of S-type). But it's not immediately clear how it relates to suffix 6.

4.1 Induced Sorting

The above gives us an $O(n)$ time method to get the correct order of the L-type suffixes given the ordering of the LMS suffixes. But two problems remain. Of course, first of all, we need to also order the S-type suffixes. And second of all, the above method doesn't really work as-is. Maintaining this tree-like structure is a HUGE pain—we were promised that we would just be working with arrays! And we haven't talked about how to "jump" to the right position in the list (i.e. how do we insert 9 after all L-type suffixes starting with C and before all S-type suffixes starting with C?). This would require even more bookkeeping.

Here is the first magical part of the SA-IS algorithm. Notice that the right-pointing edges in Figure 7 come immediately from string S and the type of each suffix. Let's run exactly the same algorithm—but now we *won't keep track of anything but the final suffix array.* This means we can't place suffixes in the rightmost column, as we did in (say) Figure 8. Instead, we'll take advantage of the fact that they're placed greedily to

put them in the correct place in the suffix array as they come.

Warmup. Let's say we want to get rid of *some* of the pointers in, say, Figure 8. Can we maintain the column of nodes on the right just using an array (without using pointers)?

With our knowledge of the string, and keeping in mind how new suffixes are placed, we can! Let's maintain the right column in an array of size n, partitioned into "buckets" for each character (using the character counts). We can begin by putting the sorted LMS suffixes into our array. If an LMS suffix begins with a character c, we're only going to put other suffixes beginning with c later in the array—so let's put all the LMS suffixes at the end of each bucket.

Now, when filling in our L-type suffixes, we always put each new L-type suffix after (i.e. towards the end) of each bucket compared to previous L-type suffixes. This is the same as filling the buckets in from left to right. Doing this, we always put each new string before all the S-type suffixes with the same first character (the LMS suffixes which we guaranteed are pushed to the end), and after all L-type suffixes with the same first character.

With this, we can maintain the right column in our example sort—all we need to do is keep track of the beginning of each bucket as new strings are inserted (easy to do with an array that's linear in the size of the alphabet). But, we still need to maintain all the right-pointing edges in our figures (e.g. in Figure 8).

Here's the magic: we don't maintain them at all. In fact we don't do *anything else*. We just iterate through the final suffix array, placing elements as we come across them. This is called the *induced sort*. Let's talk more specifically about how this works.

The induced sorting algorithm. Here is the induced sorting algorithm for the LMS suffixes and the L-type suffixes. Create a suffix array `SA`, initializing all array elements to -1. Calculate where the "bucket" for each letter begins and ends. Then, place the sorted LMS suffixes, in sorted order, as far to the right as possible in their bucket.

We then iterate through `SA` (not S!) from left to right. Each time we come across an empty slot (i.e. a slot equal to -1), we ignore it. But, each time we find a slot with an entry larger than 0, we check the type of its predecessor suffix. If it's of type L, we place that predecessor in the correct slot.

Let's create an array `C` storing the counts of each character: so `C[i]` is the number of times character `i` appears in S. We can use `C` to initialize an array `B` storing the starting point of each bucket. Furthermore, let's create an array `t` of length n such that `t[i]` stores the type of suffix `t`. Finally, let's initialize an array `E` storing the *ending* point of each bucket.

We assume that we have an array `sortedP` that contains the LMS suffixes in sorted order, so we can place them into the correct bucket one at a time. (We're placing from the end, so we iterate over `sortedP` backwards to ensure that the suffixes wind up in sorted order in the array.)

```
InducedSortLMS:   //place the LMS suffixes
  for i from |sortedP| - 1 to 0:
     firstChar ← S[sortedP[i]]
     SA[E[firstChar]] ← sortedP[i]
     E[firstChar]--
```

Then we iterate over each `SA` index i from left to right. If `SA[i]` is nonempty, and suffix `SA[i] - 1` is of type L, then we place suffix `SA[i]-1` in the next (leftmost) empty slot in its bucket.

```
InducedSortL:   //place the L-type suffixes
  for i from 0 to n-1:
    if SA[i] > 0 and t[SA[i]-1] = L:
     prevIndex ← SA[i] - 1
     firstChar ← S[prevIndex]
     SA[B[firstChar]] ← prevIndex
     B[firstChar]++
```

Intuition: We are iterating through the suffixes in exactly the same order we did before. Each time we come across a new suffix, it's placed after all L-type suffixes in its bucket (which are shunted towards the left), and before all S-type suffixes in its bucket (which are shunted towards the right).

It seems that this idea makes some sense. Can we prove that this algorithm works? We'll prove it in two parts. First, we'll show that every time an index is placed in the above, it is placed in the correct position in the final suffix array relative to the other strings in the suffix array. Second, we'll show that all L-type suffixes are placed. The proofs themselves are only briefly summarized—a formal proof would proceed by induction.

Lemma 2. *The L-type and LMS suffixes written to the array by* `InducedSortL` *are in sorted order.*

Proof sketch. This proof is by induction on the number of L-type suffixes written so far. Base case: since the LMS suffixes are in sorted order, all suffixes are in sorted order if no L-type suffixes have been written.

Our inductive hypothesis is that after k L-type suffixes have been written by `InducedSortL`, all suffixes that have been written are in sorted order.

Assuming the inductive hypothesis, we want to show that writing the $k+1$st L-type suffix retains this sorted order. Let's say the $k+1$st L-type suffix begins with character c. The only suffixes after the $k+1$st L-type suffix either are S-type and start with c, or start with a character after c. The suffixes before the $k+1$st L-type suffix either start with a character before c, or start with c and are an L-type suffix. Consider such an L-type suffix j; it must have the form $c \cdot s_j$, whereas the $k+1$st L-type suffix must have the form $c \cdot s_{k+1}$. By the induced sort algorithm, the index pointing at s_j in `SA` must have occurred before the index pointing at s_{k+1} in `SA`. Since both were in `SA`, both were among the first k suffixes written to `SA`. Therefore, by the inductive hypothesis, s_j must come before s_{k+1} in sorted order, and suffix $k+1$ is placed in the correct spot. □

Lemma 3. *All L-type suffixes are written to the array by* `InducedSortL`*.*

Proof Sketch. This proof is by induction on the number of characters between an L-type suffix and the next LMS suffix.

The inductive hypothesis is: all L-type suffixes i with at most k characters between i and the next LMS suffix are written to `SA` during `InducedSortL`.

The base case is $k = 0$. Any L-type suffix j that immediately proceeds an LMS suffix is written if $j+1$ is stored in `SA`. Since $j+1$ is an LMS suffix, $j+1$ is stored in `SA`, and j is written to `SA`.

The inductive step is essentially the same argument: for an L-type suffix j, we want to show that suffix $j+1$ is written to the `SA`; however, we must now additionally show that $j+1$ must be written to a *later* slot. (Otherwise, since we only loop over `SA` once, $j+1$ will never be considered.) However, since $j+1$ is L-type, it is lexicographically smaller than suffix j; by Lemma 2 it is written later in `SA`. □

Finally, we need to write the S-type suffixes. The key observation here is that now that we have an ordering of the L-type suffixes, we can draw a diagram very similar to Figure 7—but now, the right spine consists of the L-type suffixes, and edge dependencies show that one node is *less* than another. Note that all suffixes are now in this diagram.

With this in mind, we can continue largely as before, but in reverse: at each point in time we can correctly place the *largest* remaining suffix. Let's do one diagram of what this ordering looks like before we discuss how to use an Induced Sort.

The main rules still apply: the L-type suffixes (and LMS suffixes) are already sorted. Each new S-type suffix should appear in the bucket after all L-type suffixes with the same character, but before every other S-type suffix that was already written by the algorithm.

So our algorithm can proceed largely as before. We already have the L-type suffixes placed correctly in SA. First, we must *recalculate* E, the ends of our buckets.

What do we do about the LMS suffixes that are in SA? Figure 10 would seem to indicate that we should delete them and start from scratch. But it turns out we don't have to. In our induced sort, any time we look at an index i such that $SA[i] - 1$ is of type L (i.e. the LMS suffixes), we just skip past the index. So

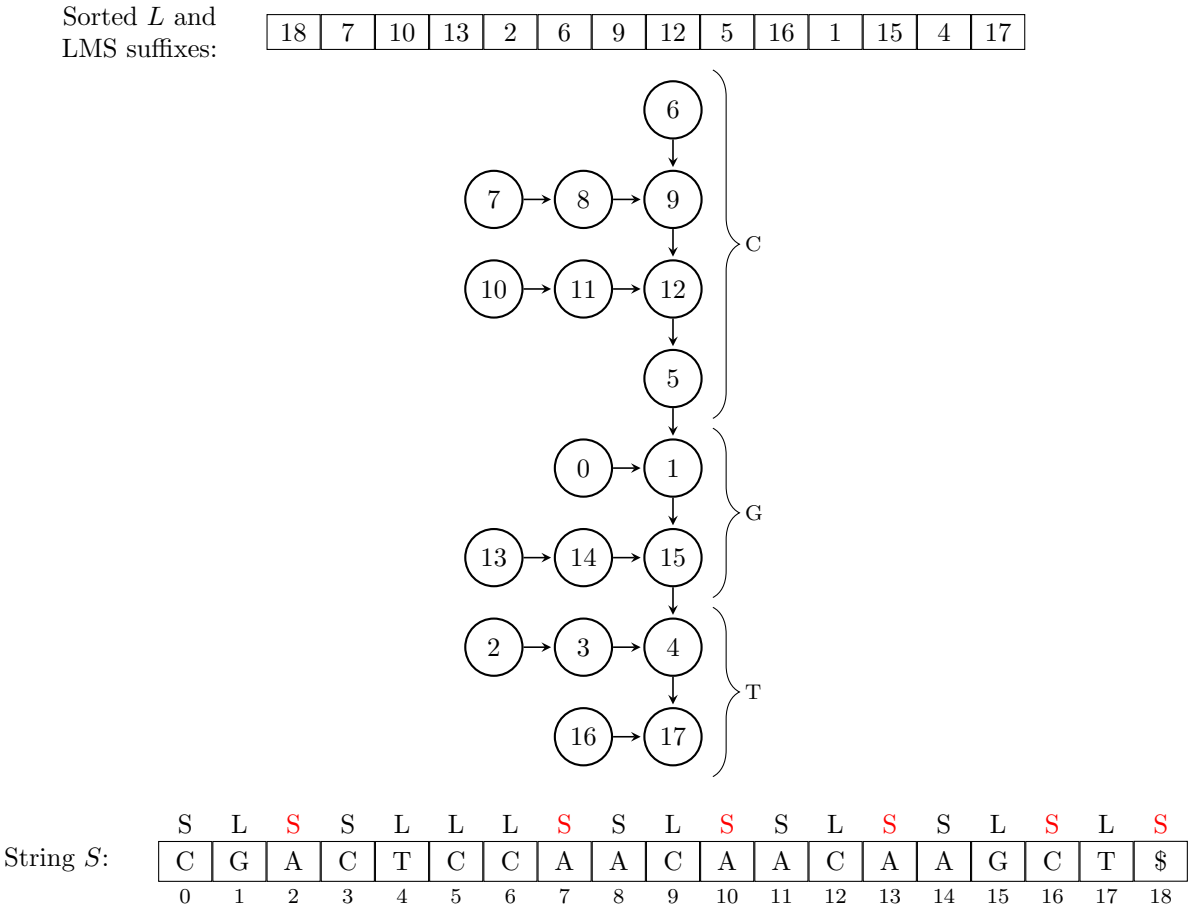


Figure 10: We know each S-type suffix is smaller than the suffix that comes after it; and, we have all the L-type suffixes (and the terminal suffix) sorted.[6] We can represent all of these dependencies graphically as above. (There is an arrow from suffix i to suffix j if i comes *before* j in sorted order.) The rightmost column of nodes consists of the L-type and LMS suffixes. All suffixes appear in some node above.

the LMS suffixes are just ignored by our algorithm—all of them will be eventually rewritten as we traverse the array.

```
InducedSortS:   //place the S suffixes
  for i from n-1 to 0:
    if SA[i] > 0 and t[SA[i]-1] = S:
     firstChar ← S[SA[i] - 1]
     SA[E[firstChar]] ← SA[i] - 1
     E[firstChar]--
```

Final Induced Sort Algorithm. To perform an induced sort, we run each of these steps one after the other. First, we calculate the counts for all buckets. We use this to calculate the endpoints of all buckets; with these endpoints we can place the LMS suffixes in the buckets. Then, we do a scan through SA to place all of the L-type suffixes. Finally, we **reset the endpoints of the buckets** and scan through SA backwards to place all S-type suffixes.

Our final algorithm looks like this. Assume (for now) that `sortedP` contains the LMS suffixes in sorted order, and there is an array C containing the counts of each character.

```
InducedSort:
  store the ends of each bucket in E using C
  store the beginning of each bucket in B using C
  for i from 0 to |sortedP| - 1:
    firstChar ← S[sortedP[i]]
    SA[E[firstChar]] = sortedP[i]
    E[firstChar]--
  for i from 0 to n-1:
    if SA[i] > -1 and t[SA[i]-1] = L:
      prevIndex ← SA[i] - 1
      firstChar ← S[prevIndex]
      SA[B[firstChar]] ← prevIndex
      B[firstChar]++
  store the ends of each bucket in E using C
  for i from n-1 to 0:
    if SA[i] > -1 and t[SA[i]-1] = S:
      prevIndex ← SA[i] - 1
      firstChar ← S[prevIndex]
      SA[E[firstChar]] = sortedP[i]
      E[firstChar]--
```

The following lemma completes the proof of correctness. The proof is not included here; it essentially combines the proofs of Lemmas 2 and 3.

Lemma 4. *If `sortedP` contains the indices of all LMS suffixes in sorted order, then after `InducedSort` is run, the suffix array of S is stored in `SA`.*

An example induced sort run. Let's look at what happens when we run induced sort on our string `CGACTCCAACAACAAGCT$`.

First, we go through the string and label each suffix as S-type or L-type, storing the result in an array `t`.

To start, we calculate our array of character counts C: there is 1 occurrence of `$`, 7 occurrences of `A`, 7 occurrences of `C`, 2 occurrences of `G`, and 2 occurrences of `T`.

Using that, we can calculate the `B` and `E` arrays. This can be done by summing over `C`: for example, there is 1 character before A so bucket A starts at position 1; there are 15 characters before G so bucket G starts at position 1. The end of each bucket is one less than the beginning of the next bucket (the end of the last bucket is the last character). (Note that I'm assuming for space that the alphabet is exactly 5 characters—I'm assuming that the alphabet is $0, 1, 2, 3, 4$ rather than `$, A, C, G, T`. You do *not* want to make this assumption in your code; you should always have `B` and `E` at least as large as your alphabet.)

```
B: 0 1 8 15 17
E: 0 7 14 16 18
```

Furthermore, we are *given* (for now!) an array `sortedP` containing pointers to the LMS suffixes in sorted order. In our case, that array is:

```
sortedP: 18 7 10 13 2 16
```

We begin by initializing array `SA` to be all -1. Then, we scan through `sortedP` and use `E` to place all LMS suffixes at the end of their bucket.

For example, we start with the last entry of `sortedP`, 16. We see that `S[16]` is `C`; therefore, we put 16 at slot 14 and decrement `E[2]`. Then we consider 2; `S[2]` is A, so we put 2 at slot 7 and decrement `E[1]`. Continuing, we obtain Figure 11 (note that `sortedP` is placed in order in `SA`).

The LMS suffixes are placed. Let's place the L-type suffixes. We scan through `SA`; each time we see an entry that is not -1, we look at the index before that entry. If it is L-type, we place it at the beginning of its bucket.

SA:	18	-1	-1	-1	7	10	13	2	-1	-1	-1	-1	-1	-1	16	-1	-1	-1	-1
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

	S	L	S	S	L	L	L	S	S	L	S	S	L	S	S	L	S	L	S
String S:	C	G	A	C	T	C	C	A	A	C	A	A	C	A	A	G	C	T	$
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Figure 11: Suffix array is initialized to all -1, and the entries of `sortedP` are placed at the end of their respective buckets.

So we start with 18; we look at suffix 17. Since it's L-type, we place it at the beginning of bucket T, in slot 17. Slots 1 through 3 are -1, so we skip them. Slot 4 contains 7, so we look at suffix 6; it is L-type, so we place it at the beginning of bucket C in slot 8. Scanning through slots 5 through 8, we place suffixes 9, 12, 1, and 5. At this point, we are considering slot 9 in `SA`:

										i ↓									
SA:	18	-1	-1	-1	7	10	13	2	6	9	12	5	-1	-1	16	1	-1	17	-1
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

	S	L	S	S	L	L	L	S	S	L	S	S	L	S	S	L	S	L	S
String S:	C	G	A	C	T	C	C	A	A	C	A	A	C	A	A	G	C	T	$
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Figure 12: We are halfway through our scan through SA placing L-type suffixes.

Subtracting one from suffix 9 we obtain suffix 8; it is S-type so we do nothing. We look at slot 10 of `SA` and see that suffix 12 is stored; suffix 12 is preceded by suffix 11; suffix 11 is S-type so we do nothing. Slot 11 contains suffix 5; it is preceded by suffix 4 which is L-type so we place it in the correct position using `B`. Slot 12 contains 4; suffix 3 is S-type so it is skipped. Slot 13 is -1 so is skipped. In slot 14 we see 16; its predecessor is suffix 15 which is L-type so we place suffix 15. Slots 15 and 16 contain suffixes that are preceded by S-type suffixes so they are skipped.

SA:	18	-1	-1	-1	7	10	13	2	6	9	12	5	-1	-1	16	1	15	17	4
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

	S	L	S	S	L	L	L	S	S	L	S	S	L	S	S	L	S	L	S
String S:	C	G	A	C	T	C	C	A	A	C	A	A	C	A	A	G	C	T	$
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Figure 13: All L-type suffixes have been placed in the array in sorted order.

Now we have placed all the L-type suffixes in the array. Looking back to Figure 6, we can see that the L-type suffixes are all in fact in the correct bucket.

Now, it's time to place the S-type suffixes. We **reset E** to point to the end of each bucket using C. We then scan through `SA` **backwards**; each time we see a suffix, we check if its predecessor is S-type; if so we

write it.

We start with slot 18; we see suffix 4. Suffix 3 is S-type, so we write it to the last slot in bucket $S[3] = C$. Recall that we reset the ends of the buckets, so bucket C now ends at 14. We write suffix 3 to slot 14, overwriting 16.

For the sake of space I won't write out the intermediate points of this algorithm, but I encourage you to do so at home. Note that we don't delete the LMS suffixes from SA in Figure 13—but we overwrite every one of them![7]

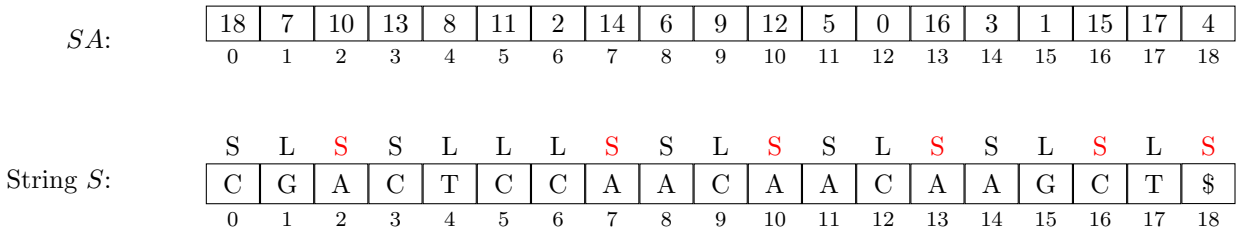


Figure 14: All suffixes have been placed in the array in sorted order!

One can verify that SA, in fact, correctly stores the suffix array of S.

Recap: We have gone over the *last step* of the algorithm. If we assume that the LMS suffixes are stored, in sorted order, in an array `sortedP`, then we can use the `InducedSort` method to complete the rest of the suffix array. This method consists of one loop over `sortedP` and two loops over `SA`, so it runs in $O(n)$ time.

5 Recursively Sorting the LMS Suffixes

Our only task now is to sort the LMS suffixes. If we can do that, we are done.

We can obtain, in $O(n)$ time, the *indices* of the LMS suffixes (not in sorted order): we label the types of the suffixes of S, and then find the LMS suffixes using a linear scan. Let's say we store them in an array `P`. So, rephrasing, our goal is to sort the suffixes stored in `P` to obtain `sortedP`.

5.1 Definitions for Building the Recursive String

I've been referring to sorting the LMS suffixes as something we're going to solve "recursively". But this isn't a recursive problem as-is. The SA-IS algorithm sorts *all* suffixes of a string S of length n. Currently, we have a *subset* of these suffixes to sort. What we want instead is to create a string S' of length $\leq n/2$, such that sorting all suffixes of S' will give us the sorted order of LMS suffixes. This is the goal of this section.

Let's write out the LMS suffixes in the order they appear in S; doing so obtains Figure 15. (Since we are writing them out in their order in S, each successive LMS suffix is a suffix of the previous LMS suffix).

To reiterate, these aren't the suffixes of a single string S'. But they are suffixes of each other. The real problem here is that there are multiple characters "between" each suffix. For example, the LMS suffix `CT$` is not followed by `T$`; instead it's followed by `$`. We'd like to *group these characters together* to form new characters. See Figure 16.

What do we need out of these new characters? We want to make sure we preserve the sorted order of the previous groups of characters. So here is our goal: group these characters together into new characters, preserving the relative order of the characters we group.

[7] To be clear: they're all overwritten, but an LMS suffix may be overwritten by itself. This does not happen in this example, however.

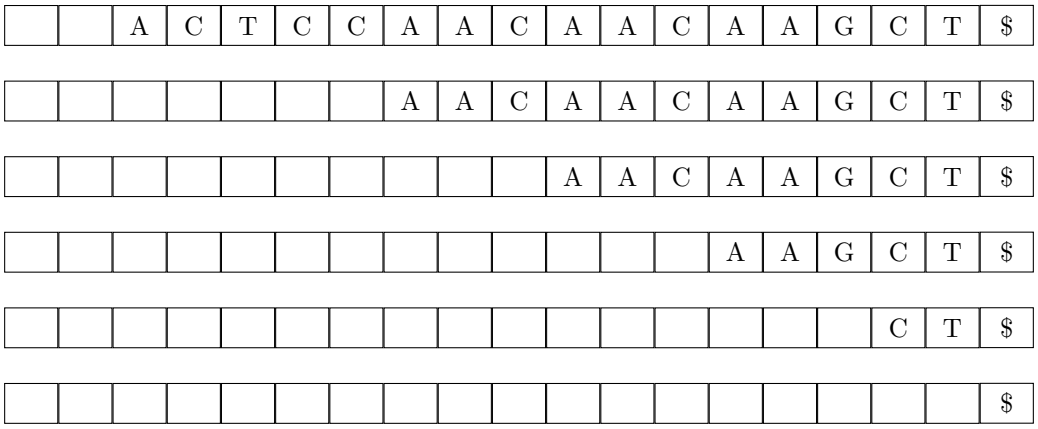


Figure 15: Listing out the LMS suffixes.

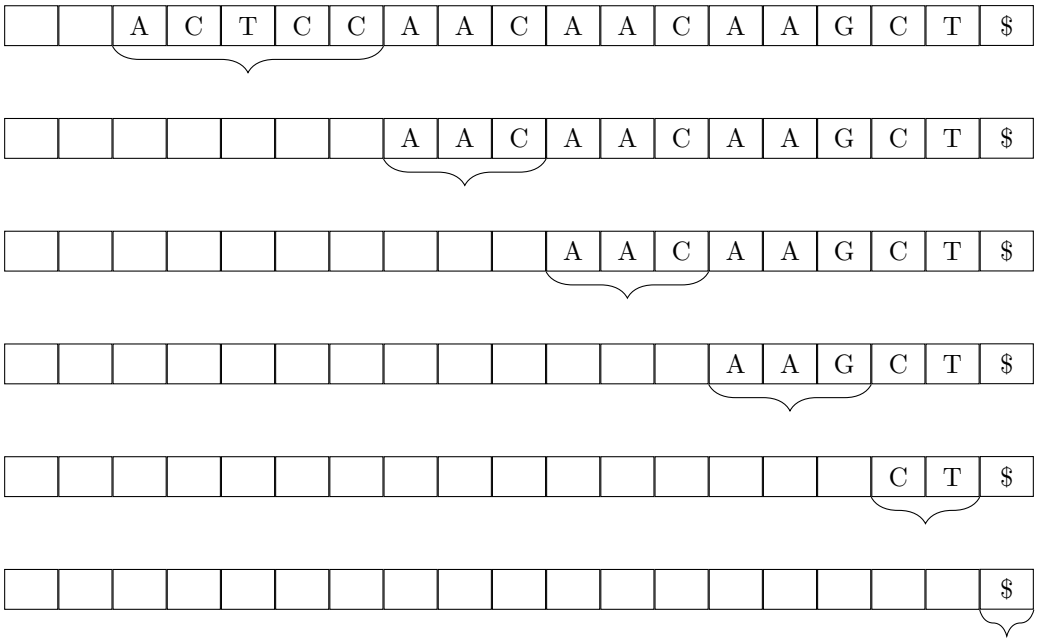


Figure 16: Our goal when making S' is to group these characters together, so that each group of characters under a brace becomes a single character. Note that when we formalize this idea into blocks, we will extend each of these groups one more character so that they overlap.

LMS Blocks. Let's take a closer look at what these "groups" really are. Each "group" starts at the beginning of an LMS suffix, and ends at the beginning of the next LMS suffix.

We define a *LMS block* to be a substring of S from i to j (i.e. $S[i], S[i+1], \ldots, S[j]$) where i and j are both LMS suffixes. The final character of S, by itself, is also considered to be an LMS block. **Note that the blocks, as defined, overlap.**

Listing out the blocks from Figure 16, we obtain the following blocks: `ACTCCA`, `AACA`, `AACA`, `AAGC`, `CT$`, `$`.

Each LMS suffix starts with some LMS block. Note that multiple LMS suffixes may start with the same LMS block (in fact, we see that LMS suffixes 7 and 10 both have LMS block `AACA`).

Assigning New Characters to LMS Blocks. As stated, we want each LMS block to be assigned some new character. Let's keep it simple and have our alphabet be $0, \ldots X$ for some X. We want the LMS blocks to be assigned to characters *in sorted order*. That means we need to sort the LMS blocks!

5.2 Sorting the LMS Blocks

Here is the true magic of the SA-IS algorithm. We want to obtain the sorted order of the LMS blocks.

The way we do that is to run the induced sort algorithm on S *without assuming the LMS suffixes are sorted*. In other words, we run the induced sort—but for `sortedP`, we just use `P` in unsorted order.

To be clear:

- We saw earlier that if our array of LMS indices `sortedP` is sorted, then an induced sort on S using `sortedP` gives us the correct suffix array.
- I am now claiming that if we just use `sortedP = P`, then an induced sort on S using `sortedP` will result in the LMS suffixes winding up in the sorted order of their LMS block.

To take a step back, each recursive call of the SA-IS algorithm is going to use two induced sorts. First, it uses an induced sort to get the LMS blocks in sorted order, to prepare for a recursive call. The recursive call will make sure that the LMS suffixes (not just the LMS blocks) are entirely sorted. Once the recursive call completes, the LMS suffixes will be in sorted order, and a new induced sort will give the correct suffix array.

Before we briefly discuss why this works, let's see a quick example of this. To reiterate, our goal is to sort the LMS blocks `ACTCCA`, `AACA`, `AACA`, `AAGC`, `CT$`, `$`. These correspond to the LMS suffixes: 2, 7, 10, 13, 16, 18.

Let's put these suffixes in `P`, and then run an induced sort To emphasize that it is not sorted, I will call this array `P`; however, it is used exactly as `sortedP` was in Section 4. We have:

```
B: 0 1 8 15 17
E: 0 7 14 16 18
P: 2 7 10 13 16 18
```

Now we run our induced sort on this "wrong" `P` exactly as we did before. Placing the LMS suffixes in `P` as if they were sorted, we obtain Figure 17.

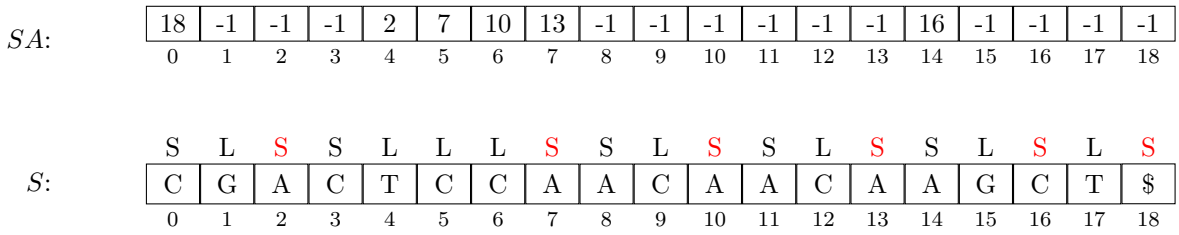


Figure 17: Suffix array is initialized to all -1, and the entries of `P` are placed at the end of their respective buckets (even though `P` is not correctly sorted).

We can immediately notice that things actually aren't so bad. When we place the LMS suffixes, they are in the correct position according to their first character. So we can state the following invariant. This invariant is pretty simple, but it's going to turn out to be very helpful when placing the L-type suffixes.

Invariant 5. *The LMS suffixes of S are placed in SA in correct sorted order according to their first character.*

Let's continue with the induced sort. In the first iteration through SA, we place the L-type suffixes.

Let's talk a little bit about how this differs from the induced sort when `P` is actually sorted. We can draw a graph just like before. But now, all we know about the relationship between the LMS suffixes is that their first characters are correctly sorted.

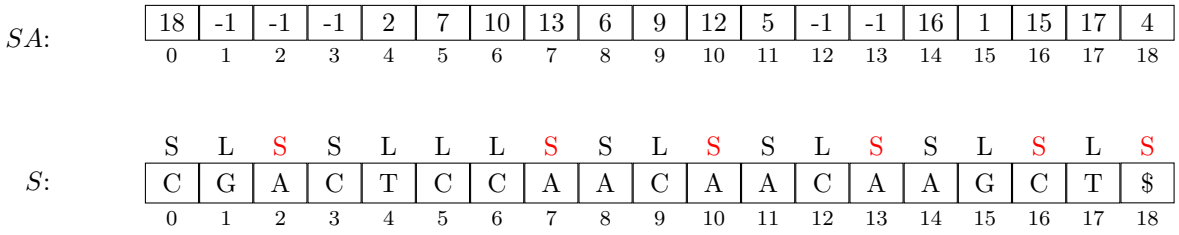


Figure 18: The L-type suffixes are placed according to the (unsorted) ordering of P.

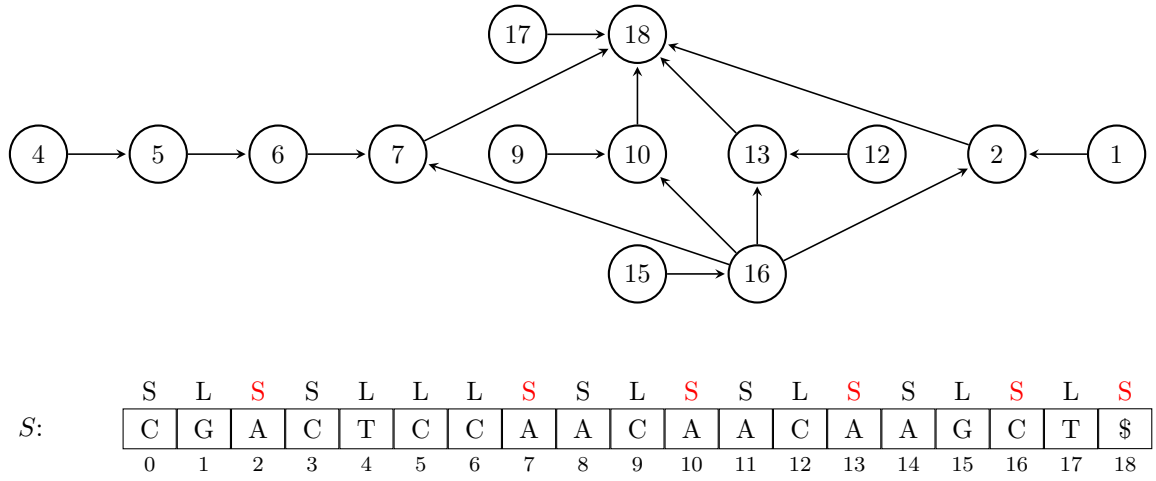


Figure 19: We know each L-type suffix is larger than the suffix that comes after it. But for LMS suffixes, all we know is that each is smaller than any LMS suffix that starts with a previous character.

We can extend the idea from the previous LMS sorting algorithm—each time we place a suffix i, it's in the correct place based on suffix $i+1$. But, of course, the LMS suffixes are only sorted up to their first character—so if suffix $i+1$ is an LMS suffix, all we know is that *that character* is correct. Putting this together, we obtain the following invariant. (The proof of this essentially just combines the proofs of Lemmas 2 and 3, modified to use Invariant 6 for the ordering of the LMS suffixes rather than assuming sorted order.)

Invariant 6. *Each L-type suffix is placed in SA in sorted order,* ***up to the next LMS suffix character****.*

In fact, looking at the result of placing the L-type suffixes (in Figure 18), we can see that this is the case. Let's look at one pair of L-type suffixes as an example. Suffix 6 is `CAAC...`; suffix 12 is `CAAG...`. Suffix 6 winds up earlier than suffix 12, even though it's later in sorted order. But the second character of each of these suffixes is from an LMS suffix—so Invariant 6 only compares `CA` from suffix 6 to `CA` from suffix 12. So Invariant 6 is satisfied for this pair.

Now, finally, we place the S-type suffixes as before (remember to reset `E` beforehand). This gives us the SA in Figure 20.

We know from Invariant 6 that the L-type suffixes are correct up to the next LMS suffix character. The S-type suffixes are sorted correctly based on the ordering of the L-type suffixes. That means that (using essentially the proof of Lemma 4) we immediately obtain:

Invariant 7. *Each S-type suffix is placed in SA in sorted order,* ***up to the next LMS suffix character****.*

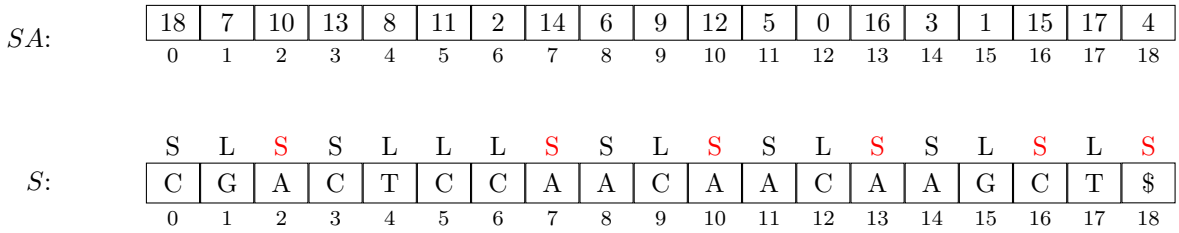


Figure 20: The S-type suffixes are placed according to the L-type suffixes.

Every LMS block begins and ends with an S-type suffix. So Invariant 7 immediately implies that the blocks are in sorted order (Corollary 8).

Corollary 8. *Let i and j be two LMS suffixes, and let B_i and B_j be the LMS blocks starting at i and j respectively. Then if i appears before j in SA, then either B_i is before B_j in sorted order, or $B_i = B_j$.*

Remember that our goal is to sort the LMS blocks. Let's say I want to create an array `sortedLMS` that contains the indices of the LMS blocks in sorted order. Then we can create `sortedLMS` for a string S in $O(n)$ time as follows. Note that we can test if a suffix $i > 0$ is an LMS suffix by checking if i is of type S, and $i - 1$ is of type L.

The following pseudocode uses this approach to create the array `sortedLMS`. We'll also create the array `P` of the LMS suffixes in the order they appeared in S (it will come in handy in Section 5.3).

```
createPandSortedLMS:
   store the types of each suffix in an array t
   create an array P of size n
   lengthOfP ← 0
   for i = 1 to n-1:
      if i is an LMS suffix:
         P[lengthOfP] = i
         lengthOfP++
   SA ← InducedSort(S,P)
   create an array sortedLMS of size lengthOfP
   j ← 0
   for i = 0 to n-1:
      if SA[i] is an LMS suffix:
         sortedLMS[j] = SA[i]
         j++
```

To complete our example, let's iterate through SA in Figure 20 (after the inductive sort was completed). We find that suffix 18 is an LMS suffix, as are suffixes 7, 10, and 13; but suffix 8 is not an LMS suffix. We continue, writing the results, to obtain

`sortedLMS: 18 7 10 13 2 16`

This corresponds to LMS blocks `$`, `AACA`, `AACA`, `AAGC`, `ACTCCA`, `CT$`, which are in fact in sorted order.

5.3 Using Sorted LMS Blocks to Make the Recursive String

From Section 5.2, we have an array `sortedLMS` that contains pointers to all LMS blocks, in sorted order. Recall that our goal is to assign LMS blocks to new characters while retaining their sorted order.

We can do this by scanning through the (now-sorted) blocks. We assign the first block in `sortedLMS` to be character 0. Then we iterate through each LMS block. If the next LMS block is the same as the previous, it is mapped to the same character. Otherwise, it is mapped to the next character.

When we are comparing characters for LMS blocks, we must compare both the character and its type. In other words, when we are comparing LMS blocks, we cannot just compare the characters at their indices; if the characters match we must also compare their types. If the types are different, they are not the same character: an L-type `A` is before an S-type `A`. (If the characters don't match, comparing types is unnecessary.)

Note that the *total size* of all LMS blocks is $O(n)$, since each character of S is in at most two LMS blocks. That means that we can compare each LMS block to the successive LMS block character by character, all in $O(n)$ worst case time.

Let's look at an example of how this works. Let's create a new array `blockAssignments` to determine the new character to assign to each block. We want to create this assignment using our array:

`sortedLMS: 18 7 10 13 2 16`

We start by setting `blockAssignments[0] = 0`. Now, we want to compare the 0th LMS block to the 1st LMS block.

We can see that $S[18] < S[7]$. That means the second block is after the first block in sorted order. We can set `blockAssignments[1] = 1`. Now, we want to compare 1st LMS block to the 2nd LMS block.

$S[7] = S[10]$ and suffixes 7 and 10 are both S-type, so the first characters match. We compare the next characters: again, $S[8] = S[11]$ and the types match. $S[9] = S[12]$ and both are L-type. $S[10] = S[13]$ and both are S-type. However, note that index 10 and index 13 are both LMS characters; that means we have finished comparing the LMS blocks. We didn't find a mismatch, so the 1st and 2nd LMS block are equal. Therefore, `blockAssignments[2] = 1`.

Now, we compare the 2nd and 3rd LMS block. Index 10 matches index 13 in character and type; same with 11 vs 14. But $S[12] = C$ whereas $S[15] = G$. Therefore, these LMS blocks are not the same. We set `blockAssignments[3] = 2`.

We continue with the comparisons, finally obtaining

`blockAssignments:  0 1 1 2 3 4`

Let's look at some pseudocode to build `blockAssignments`. When implementing this be sure to do *all* comparisons—the last character of each LMS block (i.e. the next LMS suffix character) should be compared.

```
EqualBlocks(i,j):
   c ← 0
   while c + i < n and c + j < n :
      if S[i+c] != S[j+c]:
         return false
      if t[i+c] != t[j+c]:
         return false
      if c > 1 and index i+c or j+c is an LMS suffix:
         return true
      c++

BuildBlockAssignments:
   blockAssignments[0] ← 0
   for i = 1 to |blockAssignments| - 1:
      if EqualBlocks(sortedLMS[i-1], sortedLMS[i]):
         blockAssignments[i] ← blockAssignments[i-1]
      else:
         blockAssignments[i] ← blockAssignments[i-1] + 1
```

With this array, we are ready to construct S'. Recall that our goal was to create S' so that the suffix array of S' corresponds to the sorted order of the LMS suffixes. With our assignments to LMS blocks, we can fill in how we want to substitute these characters in Figure 21.

The sorted order of the LMS suffixes (at indices 2, 7, 10, 13, 16, 18) is exactly the same as the sorted order of the following strings (the indices are written for each string so that we can keep track of where we are):

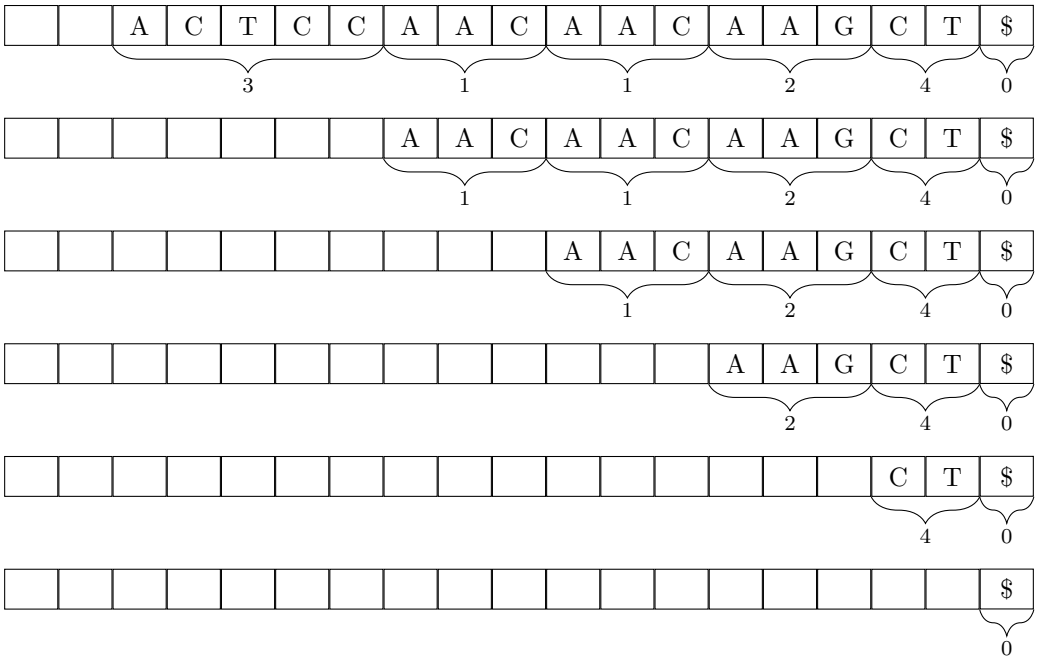


Figure 21: Using `blockAssignments`, we can substitute each of these blocks with a single character while preserving their relative ordering.

```
2:  311240
7:  11240
10:  1240
13:  240
16:  40
18:  0
```

These are the suffixes of $S' = 311240$. Let's discuss how to create this string.

Currently, `blockAssignments[i]` holds the character that we want to use for block `sortedLMS[i]`. The ith character of S', however, should be the character for block `p[i]`.

Here's one way to do this assignment. (There may be better ways, but make sure that any method is $O(n)$ time.) Create an array of length n called `pAssignments`. Then, iterate through `blockAssignments`, setting `pAssignments[sortedLMS[i]] = blockAssignments[i]`; after this, if suffix j is assigned to character c in `blockAssignments`, we will have `pAssignments[j] = c`. Then we can iterate through `P`, assigning `S'[i] = pAssignments[P[i]]`.

The pseudocode for this is as follows.

```
CreateS':
  create an array pAssignments of length n
  for i = 0 to |blockAssignments| - 1:
    pAssignments[sortedLMS[i]] ← blockAssignments[i]
  create an array S' of length |P|
  for i = 0 to |p| - 1:
    S'[i] ← pAssignments[P[i]]
```

5.4 Making the Recursive Call and Calculating `sortedP`

Somewhat unusually, the base case of the SA-IS algorithm occurs here: if S' has a special structure, we do not make a recursive call.

In particular, note that if all characters in S' are distinct, then computing the suffix array of S' is easy. Any two suffixes start with a different character, so sorting the characters gives the suffix array. Here, we have an even stronger property: the characters of S' are $0, 1, \ldots |S'| - 1$. These are trivial to sort—each character already gives its position in sorted order! In other words,

Lemma 9. *If there are $|S'|$ characters $0, 1, \ldots, |S'| - 1$ in our recursive string S', then if SA' is the suffix array of S', we have $SA'[S'[i]] = i$ for all i.*

Therefore, if the last entry of `blockAssignments` is one less than the size of `blockAssignments`, we can calculate SA' using a linear scan.

Otherwise, we do need to make a recursive call. Recall that we wanted to guarantee that $|S'| \leq |S|/2$. Let's prove this. $|S'|$ is the number of LMS suffixes in S. Each LMS suffix is an S suffix preceded by an L suffix. Therefore, there must be at least as many L suffixes as there are LMS suffixes—so $|S'| \leq |S|/2$.

Next, there's a subtle issue: we assumed that for all strings we use, the last character is the first character in sorted order (and it is the only instance of that character). Can we guarantee that S' always ends with 0, and that there is only one instance of the character 0?

The last character of S is an LMS block by definition; this LMS block must consist of the terminal character of S. This means that this final LMS block is the first LMS block in sorted order; furthermore, since there is only one instance of the terminal character in S, there can only be one instance of this LMS block. This means that S' always ends in 0, and there are no other instances of 0 in S'.

Once we get the suffix array for S', we are almost done. S' consists of the indices of the LMS blocks in the order they appear in S—in other words, the ith character in S' corresponds to the ith entry in `P`. Therefore, the suffix array for S' *gives us the sorted ordering for `P`*. We rearrange `P` using the suffix array for S' to obtain `sortedP`.

Let `SA'` be the suffix array for `S'`.

```
createSortedP:
   create an array sortedP of length |p|
   for i = 0 to |p| - 1:
      sortedP[i] = P[SA'[i]]
```

Now that we have `sortedP`, we can use an induced sort to obtain the suffix array, as we already saw in Section 4.

6 Putting it All Together

Let's discuss how the SA-IS algorithm works. Then, we'll put together pseudocode to create the suffix array of any string S in linear time.

The first step is to scan backwards through S to find the type of each suffix; these are stored in an array `t`. Then, another scan (forwards) through S allows us to find all of the LMS-type suffixes. We store all of these (in the order they appear in S) in an array `P`. Keep track of how many LMS suffixes there were in a variable `sizeOfP`.[8]

We run an induced sort on S and `P` to obtain an array `SA1`. We create an array `sortedLMS` with length `sizeOfP` to store the indices of the LMS suffixes sorted by their corresponding LMS Block. We fill `sortedLMS` by iterating through `SA1` left to right; each time we find an LMS suffix we add it to `sortedLMS`.

Now, we create an array `blockAssignments` of length `sizeOfP`. We set `blockAssignments[0] = 0`. Then we iterate through each entry $i > 1$ of `blockAssignments`. We compare the LMS block `sortedLMS[i]`

[8] In most of the above pseudocode we did not keep track of this variable for the sake of modularity. But, it's going to be very handy now that we're putting things together in one place.

with the LMS block `sortedLMS[i-1]` character by character. When doing this comparison we compare both the characters *and the type of the suffix*. If all characters (and types) in the LMS blocks are equal, we set `blockAssignments[i] = blockAssignments[i-1]`; if they are different then `blockAssignments[i] = blockAssignments[i-1] + 1`. Record the last character written as `biggestChar`.

Then, we create an array `pAssignments` of length n (i.e. the size of S—not the size of `P`) to hold how each suffix is going to be relabeled in S'. We iterate through each entry i of `blockAssignments`; we set `pAssignments[sortedLMS[i]]= blockAssignments[i]`.

We create a string `S'` with length `sizeOfP`. We iterate through S' one element at a time, and find the corresponding character by looking up `pAssignments[S[p[i]]`, storing the result in `S'[i]`.

Allocate an array `SA'` of length `sizeOfP` to store the suffix array of `S'`. If `biggestChar = sizeOfP - 1`, then for all `i`, set `SA'[S'[i]] = i`. Otherwise, recursively call SA-IS with string `S'`, length `sizeOfP`, and alphabet size `biggestChar`; store the result in `SA'`.

Use `SA'` to sort `P`: create an array `sortedP` with length `sizeOfP`. For each index `i` of `P`, set `sortedP[i] = P[SA[i]]`.

Finally, call the induced sort algorithm on S with sorted LMS suffixes stored in `sortedP`. This gives the final suffix array.

Let's put all the pseudocode to accomplish this together in one place.

```
InducedSort(S, t, pArray, n, sizeOfP):
   compute the counts of all characters and store in C
   store the ends of each bucket in E using C
   store the beginning of each bucket in B using C
   create an array SA of size n
   for i from sizeOfP - 1 to 0:
      firstChar ← S[pArray[i]]
      SA[E[firstChar]] ← pArray[i]
      E[firstChar]--
   for i from 0 to n-1:
      prevIndex ← SA[i] - 1
      if SA[i] > 0 and t[prevIndex] = L:
         firstChar ← S[prevIndex]
         SA[B[firstChar]] ← prevIndex
         B[firstChar]++
   store the ends of each bucket in E using C
   for i from n-1 to 0:
      prevIndex ← SA[i] - 1
      if SA[i] > 0 and t[prevIndex] = S:
         firstChar ← S[prevIndex]
         SA[E[firstChar]] ← prevIndex
         E[firstChar]--
   return SA
```

```
CreateSortedLMS(SA, t, n, sizeOfP):
   create an array sortedLMS of size sizeOfP
   j ← 0
   for i = 0 to n-1:
      if SA[i] is an LMS suffix:
         sortedLMS[j] = SA[i]
         j++
   return sortedLMS
```

```
EqualBlocks(S, t, n, i, j):
   c ← 0
   while c + i < n and c + j < n :
      if S[i+c] != S[j+c]:
         return false
      if t[i+c] != t[j+c]:
         return false
      if c > 1 and index i+c or j+c is an LMS suffix:
         return true
      c++

BuildBlockAssignments(S, t, sortedLMS, n, sizeOfP):
   create an array blockAssignments of size sizeOfP
   blockAssignments[0] ← 0
   for i = 1 to sizeOfP - 1:
      if EqualBlocks(S, t, n, sortedLMS[i-1], sortedLMS[i]):
         blockAssignments[i] ← blockAssignments[i-1]
      else:
         blockAssignments[i] ← blockAssignments[i-1] + 1
```

```
CreateS'(sortedLMS, blockAssignments, P, n, sizeOfP):
   create an array pAssignments of length n
   for i = 0 to sizeOfP - 1:
      pAssignments[sortedLMS[i]] ← blockAssignments[i]
   create an array S' of length sizeOfP
   for i = 0 to sizeOfP - 1:
      S'[i] ← pAssignments[P[i]]
   return S'
```

Finally, the main algorithm that calls the above helper methods.

```
SA-IS(S, n, numCharacters):
   store the types of each suffix in an array t
   create an array P of size n
   sizeOfP ← 0
   for i = 1 to n-1:
      if i is an LMS suffix:
         P[sizeOfP] = i
         sizeOfP++

   SA ← InducedSort(S, t, P, n, sizeOfP)

   sortedLMS ← CreateSortedLMS(SA, t, n, sizeOfP)

   blockAssignments ← BuildBlockAssignments(sortedLMS, S, n, sizeOfP, t)

   create an array SA' of size sizeOfP
   S' ← CreateS'(sortedLMS, blockAssignments, P, n, sizeOfP)
   if blockAssignments[sizeOfP - 1] = sizeOfP - 1:
      for i = 0 to sizeOfP - 1:
         SA'[S'[i]] = i
   else:
      SA' ← SA-IS(S', sizeOfP, sizeOfP)

   create an array sortedP of length sizeOfP
   for i = 0 to sizeOfP - 1:
      sortedP[i] = P[SA'[i]]

   SA ← InducedSort(S, t, sortedP, n, sizeOfP)
   return SA
```