# 451: Suffix Arrays

G. Miller, K. Sutner
Carnegie Mellon University
2020/11/24

1. Suffix Arrays

2. Applications

Suppose $W$ is a string of length $n$. We can store the suffixes of $W$ in two structures:

**Suffix Tree** Path compacted, deterministic trie of size $O(n)$. Can be constructed in linear time.

**Suffix Automaton** Minimal PDFA for $\text{suff}(W)$ of size $O(n)$. Can be constructed in linear time.

Both structures are essentially finite state machines, so we implicitly also store all the factors.

Linear is asymptotically optimal, but there is still room to improve the constants, and perhaps the difficulty of the algorithms.

Here is a computationally attractive alternative to suffix trees: so-called suffix arrays.

To construct a suffix array for $W$,

- sort all the suffixes of $W$, and
- return a list of the corresponding indices in $[n]$.

The result is the suffix array $\mathrm{SA}(W)$ of $W$. Technically, $\mathrm{SA} : [n] \to [n]$ but we will sometimes conflate the index $\mathrm{SA}(i)$ and the actual suffix $w[\mathrm{SA}(i):]$.

The brute-force approach to computing the suffix array is quadratic since the total length of all suffixes is quadratic.

As it turns out, it is convenient to augment a suffix array with another array: the longest common prefix (LCP) array.

$$\mathsf{LCP}(i) = \max\big( |z| \mid z \in \mathsf{pref}(\mathsf{SA}(i)) \cap \mathsf{pref}(\mathsf{SA}(i{+}1)) \big)$$

In other words, if $u$ and $v$ are the two consecutive suffixes in lexicographic order, we need to compute the position $k$ such that $u[1{:}k] = v[1{:}k]$ whereas $u_{k+1} \neq v_{k+1}$.

Having longest common prefixes between successive suffixes sometimes comes in handy (in conjunction with the actual suffix array). Introduced by Manber in 1993 (used in `glimpse`).

For the Fibonacci word $F_6 = abaababa$ we have

| $i$ | $w[j{:}]$ | $j = \mathsf{SA}(i)$ |
|---|---|---|
| 1 | $a$ | 8 |
| 2 | $aababa$ | 3 |
| 3 | $aba$ | 6 |
| 4 | $abaababa$ | 1 |
| 5 | $ababa$ | 4 |
| 6 | $ba$ | 7 |
| 7 | $baababa$ | 2 |
| 8 | $baba$ | 5 |

So the suffix array and LCP arrays are

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| SA | 8 | 3 | 6 | 1 | 4 | 7 | 2 | 5 |
| LCP | 1 | 1 | 3 | 3 | 0 | 2 | 2 | – |

**Idea:** Instead of sorting the suffixes completely, just sort according to the first two letters: $x \preceq y$ if $x_1 x_2 \leq y_1 y_2$.

To avoid a special case with the one-letter suffix $w[n{:}]$, we use an endmarker $\#$ considered to be less than all $a \in \Sigma$.

Now suppose we repeat: in round 2, the super-letters represent strings in $\Sigma_{\#}^4$. After $k$ rounds, the super-letters represent strings in $\Sigma_{\#}^{2^k}$, so we need at most $\log n$ rounds until the super-letters correspond to the whole suffix (there is no problem with length because we pad with endmarkers).

But then we have sorted the suffixes, period. We can easily extract the suffix array.

For simplicity we will only use digit alphabets

$$\mathcal{D}_d = \{0, 1, \ldots, d\text{-}1\}$$

Sliding a window of width 2 across a word $w$ produces a sequence of "super-letters" in the alphabet $\mathcal{D} \times \mathcal{D}$. This is called a sliding block code in symbolic dynamics.

For example, $F_6 = 01001010$ turns into $01, 10, 00, 01, 10, 01, 10$ which we could renumber as $1\,2\,0\,1\,2\,1\,2$, a string over $D_4$.

This renumbering is naturally order preserving, as far as sorting is concerned nothing changes.

The length of the code word is $|w| - 1$, The number of super-letters appearing in it is at most

$$\min(|w| - 1, |\Sigma|^2)$$

To add an endmarker to a digit alphabet we use $\overline{1}$, think of this as -1 for comparisons. If $x \in \mathcal{D}^n$ we adopt the convention that $x_i = \overline{1}$ for $i > n$.

We will use fixed length word lists of the form

$$X = (x_1, x_2, \ldots, x_n)$$

where $x_i \in \mathcal{D}^{1,2}$ (but the underlying alphabet will change). A word $x$ of length $n$ is considered to be the list $X$ of its letters.

Here are the critical operations on these lists. Let $\delta = 2^k$.

pairing produce the list

$$X' = (x_1x_{1+\delta}, x_2x_{2+\delta}, \ldots, x_nx_{n+\delta})$$

recode Sort the last list, then scan left to right and associate the two-letter words with $0, 1, \ldots, \ell$ (where $\ell \leq n$-1, incrementing whenever a new word appears. Then replace the words in $X'$ accordingly.

Initialize with the character list $X$ of $w$, $k = 0$.

**Round** $k$

$$\delta = 2^k$$
$$X = \mathsf{recode}(\mathsf{pairing}_\delta(X))$$
$$k{+}{+}$$

Keep going until the words (ultimately: letters) in $X$ are all distinct. Then read the suffix array off the positions in $X$: first form pairs with $[n]$, sort and project away the first component.

**Claim:** Worst case running time is $O(n \log n)$ (but fewer rounds may suffice).

There are 3 rounds.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 2 | 3 | 1 | 2 | 3 | 2 | 3 | 0 |
| 2 | 3 | 6 | 1 | 4 | 7 | 2 | 5 | 0 |

Form pairs with the last list and $[8]$ and sort:

$$0{:}8, 1{:}3, 2{:}6, 3{:}1, 4{:}4, 5{:}7, 6{:}2, 7{:}5$$

Projecting away the first component produces the suffix array

$$8, 3, 6, 1, 4, 7, 2, 5$$

The first round unfolds like this

$$01, 10, 00, 01, 10, 01, 10, 0\overline{1}$$

$$0\overline{1}, 00, 01, 01, 01, 10, 10, 10$$

$$0\overline{1} \mapsto 0, 00 \mapsto 1, 01 \mapsto 2, 10 \mapsto 3$$

A random word over a three-letter alphabet of length 10:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 3 | 1 | 1 | 1 | 1 | 3 | 2 | 2 | 3 | 3 |
| 1 | 5 | 0 | 0 | 0 | 1 | 6 | 2 | 3 | 7 | 4 |
| 2 | 7 | 0 | 1 | 2 | 3 | 8 | 4 | 5 | 9 | 6 |

Suffix array: $2, 3, 4, 5, 7, 8, 10, 1, 6, 9$ and corresponding suffixes:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 1 | 1 | 1 | 3 | 2 | 2 | 3 | 3 |
| 3 | 1 | 1 | 1 | 3 | 2 | 2 | 3 | 3 | |
| 4 | 1 | 1 | 3 | 2 | 2 | 3 | 3 | | |
| 5 | 1 | 3 | 2 | 2 | 3 | 3 | | | |
| 7 | 2 | 2 | 3 | 3 | | | | | |
| 8 | 2 | 3 | 3 | | | | | | |
| 10 | 3 | | | | | | | | |
| 1 | 3 | 1 | 1 | 1 | 1 | 3 | 2 | 2 | 3 | 3 |
| 6 | 3 | 2 | 2 | 3 | 3 | | | | |
| 9 | 3 | 3 | | | | | | | |

The brute-force approach to computing LCP given SA is quadratic: just compute the longest prefix of all adjacent pairs

$$\big(\mathsf{SA}(i), \mathsf{SA}(i{+}1)\big) \qquad i = 1, \ldots, n{-}1$$

**Wild Idea:** How about changing the order in which we compute the prefixes of adjacent strings?

OK, but what order should we pick? Note that SA is a permutation of $[n]$, so there is an inverse permutation $\mathsf{SA}^{-1}$. Define a weird successor function $\sigma(i) = \mathsf{SA}^{-1}(i) + 1$. So we could also do

$$\big(i, \mathsf{SA}(\sigma(i))\big) \qquad i = 1, \ldots, n^{*}$$

where the $*$ indicates that we should skip over $i_0 = \mathsf{SA}(n)$.

Why on earth should this help?

**Proposition**

*If* $k = \mathsf{LCP}\big(i, \sigma(i)\big)$, *then* $\mathsf{LCP}\big(i+1, \sigma(i+1)\big) \geq k - 1$.

*Proof.*

Write $u_1, u_2, \ldots, u_n$ for the non-empty suffixes of $w$.

Suppose the algorithm moves from $(i, j)$ to $(i', j')$ where $i = i + 1$, and $u_i, u_j$ share a longest prefix of length $k$. Wlog $k \geq 2$. Then

$$u_i = a u_{i+1} \qquad u_j = a u_{j+1} \qquad u_{i+1} < u_{j+1}$$

If $j' = j + 1$ we are done. Otherwise we must have

$$u_{i+1} < u_{j'} < u_{j+1}$$

But then $k$ can drop at most by 1.

□

We can exploit the proposition by starting the next prefix search at position $k-1$, rather than $1$ in the brute-force approach.

What is the effect on the running time?

Think of $k = 1$ initially, after a phantom round 0.

In each real round, we first decrease $k$ once, and then possibly increase it some number of times. The total decrease is $n-1$. Since $k < n$ always, the total increase can be at most $2n$. So the algorithm is linear.

Again for $F_6 = 01001010$ we have

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $\mathsf{SA}(i)$ | 8 | 3 | 6 | 1 | 4 | 7 | 2 | 5 |
| $\mathsf{SA}^{-1}(i)$ | 4 | 7 | 2 | 5 | 8 | 3 | 6 | 1 |

which produces the following LCP order:

| $i$ | $j$ | $u_i$ | $u_j$ | $LCP$ |
|---|---|---|---|---|
| 1 | 4 | 01001010 | 01010 | 3 |
| 2 | 5 | 1001010 | 1010 | 2 |
| 3 | 6 | 001010 | 010 | 1 |
| 4 | 7 | 01010 | 10 | 0 |
| 6 | 1 | 010 | 01001010 | 3 |
| 7 | 2 | 10 | 1001010 | 2 |
| 8 | 3 | 0 | 001010 | 1 |

An alternative approach is to use a rolling hash (as in Rabin-Karp) to find the suffix array in (expected) time $O(n \log^2 n)$.

The idea is to preprocess the string $w$, so that a lexicographic comparison of two suffixes can be handled in $O(\log n)$ steps: given two suffixes $u$ and $v$, we compute the longest common prefix: use binary search to find the least index $k$ such that $u[:k\text{–}1] = v[:k\text{–}1]$ whereas $u_k \neq v_k$.

**Exercise**

*Figure out the details of this algorithm.*

**Question:** Can we convert between suffix trees/automata/arrays? In linear time?

This is not meant as a practical algorithm (the direct methods are faster), but a question about the logical connections.

**Arrays and Trees**

Converting the tree to the array is fairly easy: just do a traversal in lexicographic order, essentially just DFS. This also builds the LCP array.

For the opposite direction, exploit the fact that we can insert the prefixes in lexicographic order: suppose $u$ and $v$ are two consecutive suffixes and that $u$ has just been inserted. Think about pushing pointers to the nodes on the path corresponding to $u$ on a stack.

Use the LCP array to pop the stack to get back to the fork between $u$ and $v$. Then vanilla insert the tail of $v$.

Essentially the same argument applies to converting the suffix automaton to the tree: ignoring compaction, the tree is just the unfolding of the acyclic automaton. We can run a lexicographic "DFS" in the automaton, just as in the tree. Quotation marks since this version of DFS stops at the terminal node, not when a vertex is already reached.

The same method also produces the suffix and LCP array.

Getting the suffix automaton runs into problems: we could interpret the suffix tree as an automaton and use the fact that an acyclic automaton can be minimized in linear time, but the edge labels are in $\Sigma^\star$, not in $\Sigma$. Alas, in general we cannot unfold these edges in linear time. Just think about $a^n b^n \#$.

Sharing isomorphic substructures seems inherently harder.

As always, assume we have a (long) text $T$ of length $n$. Here are some standard questions regarding a potential substring $w$ of length $m$.

- Determine whether $w$ appears in $T$ (is a factor).
- Find the first position of $w$ in $T$.
- Find all the positions of $w$ in $T$.
- Count the number of occurrences of $w$ in $T$.

Suppose we have the suffix tree $\mathfrak{T}$ for $T$, which can be constructed in time linear in $|T|$. For long $T$, this is attractive mostly when we have to deal with multiple queries.

**Proposition**

*We can determine the longest prefix of $w$ that appears in $T$ in time linear in $m$.*

The number of occurrences of $w$ in $T$ is the number of leaves below the node $\nu$ in $\mathfrak{T}$ reached via $w$.

To find all positions, assume that the final nodes are labeled by the starting position of the corresponding suffix. Again we can traverse the subtree of $\nu$.

This can be handled in $O(n+m)$ steps.

**Note:** This is no better than KMP, but there is a major difference:

- preprocessing in KMP is $O(m)$, check $O(n)$,
- preprocessing for suffix tree is $O(n)$, check is $O(m)$.

Use accordingly.

Aka longest common substring (do not confuse with longest common subsequence).

**Problem:** Find a longest word in $\mathsf{fact}(T_1) \cap \mathsf{fact}(T_2)$.

Think about this for a bit. It's a simple enough question, but it is not so clear how one would go about finding an efficient algorithm.

With suffix trees, this can be handled in linear time: let $\#_1$ and $\#_2$ be two endmarkers. Let $W = T_1\#_1T_2\#_2$ and build the tree for $W$.

Clearly, the suffixes of $W$ come in two kinds: containing $\#_1$ (and a suffix of $T_1$) or not (just a suffix of $T_2$). We can label the leaves accordingly, and propagate the labels up to the internal nodes. Then find the deepest node that has both kinds in the subtree below.

The post-processing is linear time, just like the construction of the suffix tree.

Suppose we have a collection of $m$ strings $T_1, T_2, \ldots, T_m$ of total size $n$.

**Problem:** We would like to find a long string in common to many of the $T_i$.

Somewhat more precisely, for $k > 1$, define

$$\ell(k) = \max\big( |z| \mid \exists\, I \subseteq [m]\, (|I| \geq k, z \in \mathsf{fact}(T_i) \text{ for } i \in I) \big)$$

$\ell(1)$ is pointless, $\ell(2)$ we just handled.

In general, once we have $\ell(2), \ell(3), \ldots, \ell(m)$ we can try to make sense out of the informal question.

Amazingly, there is a linear time algorithm for this, but we will make do with $O(mn)$.

First, build a (generalized) suffix tree $\mathfrak{T}$ for the sequence

$$T_1\#_1,\ T_2\#_2,\ \ldots,\ T_m\#_m$$

Note that all the endmarkers are distinct.

It should be intuitively clear what we mean by a generalized suffix tree. Here is a horrible way of building it: construct the tree for the long word

$$T_1\#_1T_2\#_2\ldots T_m\#_m$$

Then delete all the subtrees below the first $\#$ on any branch starting at the root.

For any internal node $\nu$ of the tree, define

$$C(\nu) = \text{number of distinct endmarkers in the subtree of } \nu$$

The number of leaves below a node is easy to compute in linear time, but we are counting distinct leaves; this will take us $O(mn)$ steps. We compute the set of endmarkers below each node.

Assume we have the $C$ counts as well as the string-depth of all nodes in $\mathfrak{T}$. String-depth $\mathsf{sd}(\nu)$: count the number of letters on a path to $\nu$ (recall that $\mathfrak{T}$ is compacted, so this is not the same as path length).

Traverse the tree to find

$$D_k = \max\big(\, \mathsf{sd}(\nu) \mid C(\nu) = k \,\big)$$

Postprocess by setting $D_k = \max D_k, \ldots, D_m$ and we have $\ell$.

Total running time is $O(mn)$.

Suppose we have the suffix array $A$ for some text $T$ of length $n$.

**Question:** How do we actually search for a factor $w$ of length $m$?

We conduct a binary search: essentially, we maintain and shrink a range $[lo, hi]$ of possible positions for $w$ in $A$.

Initially, $lo = 1$ and $hi = n$.

Let $\ell$ be the midpoint between $lo$ and $hi$ and compare $w$ to $A(m)$, then update $lo$ or $hi$ accordingly (get out if $w$ has been found).

Total cost is $O(m \log n)$.

There are methods linear in $m$, but they are more complicated.

Suppose we have the suffix automaton $\mathfrak{A}(w)$ and $x \in \mathsf{fact}(w)$. By "position of $x$ in $w$" we always mean the position of the first letter of $x$. Let $x^{(1)}$ denote the first occurrence of $x$ in $w$. Define "first position" and "longest right context" functions as follows:

$$\mathsf{fpos}_w(x) = \text{first position of } x^{(1)}$$
$$\mathsf{lcnt}_w(x) = \max\big(\, |z| \mid z \in \mathcal{R}_w(x) \,\big)$$

Clearly $(\mathsf{fpos}_w(x) + |x| - 1) + \mathsf{lcnt}_w(x) = n$.

Let $q_{\mathsf{lst}}$ be the last state in $\mathfrak{A}(w)$. Confusing states with inputs as usual, to compute $\mathsf{lcnt}_w(x)$, exploit $\mathsf{lcnt}_w(q_{\mathsf{lst}}) = 0$ and

$$\mathsf{lcnt}_w(p) = 1 + \max\big(\, \mathsf{lcnt}_w(q) \mid q = \delta(p, a)), a \in \Sigma \,\big)$$

This can be handled with DFS in $\mathfrak{A}(w)$.

So with $O(n)$ preprocessing we can find the first position of $x$ in time $O(|x|)$.

Can we use the suffix automaton $\mathfrak{A}(w)$ to count the number of occurrences of $x \in \mathsf{fact}(w)$?

This time we need to determine the cardinality of the right context of $x$: the number of suffixes $z$ such that $xz$ is a suffix. This is similar to the last question, but now we need not just the longest context.

Let $F$ be the final states of $\mathfrak{A}(w)$, and, for any state $p$, define

$$\mathsf{cnt}(p) = |\{\, z \in \Sigma^\star \mid \delta(p, z) \in F \,\}|$$

If we can compute $\mathsf{cnt}(p)$ the answer is $\mathsf{cnt}(\delta(q_{\mathsf{ini}}, x))$.

For the computation we use

$$\mathsf{cnt}(p) = \begin{cases} 1 + \sum(\mathsf{cnt}(q) \mid q = \delta(p,a), a \in \Sigma) & \text{if } p \in F, \\ \sum(\mathsf{cnt}(q) \mid q = \delta(p,a), a \in \Sigma) & \text{otherwise.} \end{cases}$$

Again, this can be handled in a precomputation with DFS in $\mathfrak{A}(w)$ in time $O(n)$.

After the precomputation, we can count the number of occurrences of $x$ in time $O(|x|)$.