LCP Array Construction

The LCP array is easy to compute in linear time using the suffix array SA and its inverse SA^{-1}. The idea is to compute the lcp values by comparing the suffixes, but skip a prefix based on a known lower bound for the lcp value obtained using the following result.

Lemma 5.10: For any $i \in [0..n)$, $LCP[SA^{-1}[i+1]] \geq LCP[SA^{-1}[i]] - 1$

Proof. Let T_j be the lexicographic predecessor of T_i, i.e., $T_j < T_i$ and there are no other suffixes between them in the lexicographical order.

- Then $LCP[SA^{-1}[i]] = lcp(T_i, T_j) = \ell$. If $\ell = 0$, the claim is trivially true.
- If $\ell > 0$, then for some symbol c, $T_i = cT_{i+1}$ and $T_j = cT_{j+1}$. Thus $T_{j+1} < T_{i+1}$ and $lcp(T_{i+1}, T_{j+1}) = \ell - 1$.
- Let T_k be the immediate lexicographical predecessor of T_{i+1}. Then either $k = j+1$ or $T_{j+1} < T_k < T_{i+1}$. In either case,

$$LCP[[SA^{-1}[i+1]] = lcp(T_{i+1}, T_k) \geq lcp(T_{i+1}, T_{j+1}) = \ell - 1 \ .$$

$\square$

The algorithm computes the lcp values in the order that makes it easy to use the above lower bound.

Algorithm 5.11: LCP array construction
Input: text $T[0..n]$, suffix array $SA[0..n]$, inverse suffix array $SA^{-1}[0..n]$
Output: LCP array $LCP[1..n]$

(1) $\ell \leftarrow 0$
(2) for $i \leftarrow 0$ to $n-1$ do
(3) $\quad k \leftarrow SA^{-1}[i]$ $\qquad // \ i = SA[k]$
(4) $\quad j \leftarrow SA[k-1]$
(5) $\quad$ while $T[i+\ell] = T[j+\ell]$ do $\ell \leftarrow \ell + 1$
(6) $\quad LCP[k] \leftarrow \ell$
(7) $\quad$ if $\ell > 0$ then $\ell \leftarrow \ell - 1$
(8) return LCP

The time complexity is $\mathcal{O}(n)$:

- Everything except the while loop on line (5) takes clearly linear time.
- Each round in the loop increments ℓ. Since ℓ is decremented at most n times on line (7) and cannot grow larger than n, the loop is executed $\mathcal{O}(n)$ times in total.

Suffix Array Construction

Suffix array construction means simply sorting the set of all suffixes.

- Using standard sorting or string sorting the time complexity is $\Omega(dp(T_{[0..n]}))$.

- Another possibility is to first construct the suffix tree and then traverse it from left to right to collect the suffixes in lexicographical order. The time complexity is $\mathcal{O}(n)$ on a constant alphabet.

Specialized suffix array construction algorithms are a better option, though.

In fact, possibly the fastest way to construct a suffix tree is to first construct the suffix array and the LCP array, and then the suffix tree using the algorithm we saw earlier.

Prefix Doubling

Our first specialized suffix array construction algorithm is a conceptually simple algorithm achieving $\mathcal{O}(n \log n)$ time.

Let T_i^ℓ denote the text factor $T[i..\min\{i+\ell, n+1\})$ and call it an ℓ-factor. In other words:

- T_i^ℓ is the factor starting at i and of length ℓ except when the factor is cut short by the end of the text.
- T_i^ℓ is the prefix of the suffix T_i of length ℓ, or T_i when $|T_i| < \ell$.

The idea is to sort the sets $T_{[0..n]}^\ell$ for ever increasing values of ℓ.

- First sort $T_{[0..n]}^1$, which is equivalent to sorting individual characters. This can be done in $\mathcal{O}(n \log n)$ time.
- Then, for $\ell = 1, 2, 4, 8, \ldots$, use the sorted set $T_{[0..n]}^\ell$ to sort the set $T_{[0..n]}^{2\ell}$ in $\mathcal{O}(n)$ time.
- After $\mathcal{O}(\log n)$ rounds, $\ell > n$ and $T_{[0..n]}^\ell = T_{[0..n]}$, so we have sorted the set of all suffixes.

We still need to specify, how to use the order for the set $T^\ell_{[0..n]}$ to sort the set $T^{2\ell}_{[0..n]}$. The key idea is assigning order preserving names for the factors in $T^\ell_{[0..n]}$. For $i \in [0..n]$, let N^ℓ_i be an integer in the range $[0..n]$ such that, for all $i, j \in [0..n]$:

$$N^\ell_i \le N^\ell_j \text{ if and only if } T^\ell_i \le T^\ell_j \ .$$

Then, for $\ell > n$, $N^\ell_i = SA^{-1}[i]$.

For smaller values of ℓ, there can be many ways of satisfying the conditions and any one of them will do. A simple choice is

$$N^\ell_i = |\{j \in [0,n] \mid T^\ell_j < T^\ell_i\}| \ .$$

Example 5.12: Prefix doubling for $T =$ banana\$.

N^1		N^2		N^4		$N^8 = SA^{-1}$	
4	b	4	ba	4	bana	4	banana\$
1	a	2	an	3	anan	3	anana\$
5	n	5	na	6	nana	6	nana\$
1	a	2	an	2	ana\$	2	ana\$
5	n	5	na	5	na\$	5	na\$
1	a	1	a\$	1	a\$	1	a\$
0	\$	0	\$	0	\$	0	\$

Now, given N^ℓ, for the purpose of sorting, we can use

- N_i^ℓ to represent T_i^ℓ
- the pair $(N_i^\ell, N_{i+\ell}^\ell)$ to represent $T_i^{2\ell} = T_i^\ell T_{i+\ell}^\ell$.

Thus we can sort $T_{[0..n]}^{2\ell}$ by sorting pairs of integers, which can be done in $\mathcal{O}(n)$ time using LSD radix sort.

Theorem 5.13: The suffix array of a string $T[0..n]$ can be constructed in $\mathcal{O}(n \log n)$ time using prefix doubling.

- The technique of assigning order preserving names to factors whose lengths are powers of two is called the Karp–Miller–Rosenberg naming technique. It was developed for other purposes in the early seventies when suffix arrays did not exist yet.
- The best practical implementation is the Larsson–Sadakane algorithm, which uses ternary quicksort instead of LSD radix sort for sorting the pairs, but still achieves $\mathcal{O}(n \log n)$ total time.

Let us return to the first phase of the prefix doubling algorithm: assigning names N_i^1 to individual characters. This is done by sorting the characters, which is easily within the time bound $\mathcal{O}(n \log n)$, but sometimes we can do it faster:

- On an ordered alphabet, we can use ternary quicksort for time complexity $\mathcal{O}(n \log \sigma_T)$ where σ_T is the number of distinct symbols in T.
- On an integer alphabet of size n^c for any constant c, we can use LSD radix sort with radix n for time complexity $\mathcal{O}(n)$.

After this, we can replace each character $T[i]$ with N_i^1 to obtain a new string T':

- The characters of T' are integers in the range $[0..n]$.
- The character $T'[n] = 0$ is the unique, smallest symbol, i.e., $.
- The suffix arrays of T and T' are exactly the same.

Thus, we can assume that the text is like T' during the suffix array construction. After the construction, we can use either T or T' as the text depending on what we want to do.

Recursive Suffix Array Construction

Let us now describe a linear time algorithms for suffix array construction. We assume that the alphabet of the text $T[0..n)$ is $[1..n]$ and that $T[n] = 0$ (=$ in the examples).

The outline of the algorithms is:

0. Choose a subset $C \subset [0..n]$.

1. Sort the set T_C. This is done by a reduction to the suffix array construction of a string of length $|C|$, which is done recursively.

2. Sort the set $T_{[0..n]}$ using the order of T_C.

The set C can be chosen so that

- $|C| \leq \alpha n$ for a constant $\alpha < 1$.
- Excluding the recursive call, all steps can be done in linear time.

Then the total time complexity can be expressed as the recurrence $t(n) = \mathcal{O}(n) + t(\alpha n)$, whose solution is $t(n) = \mathcal{O}(n)$.

The set C must be chosen so that:

1. Sorting T_C can be reduced to suffix array construction on a text of length $|C|$.
2. Given sorted T_C the suffix array of T is easy to construct.

We look at two different ways of choosing C leading to two different algorithms:

- DC3 uses difference cover sampling
- SAIS uses induced sorting

Difference Cover Sampling

A difference cover D_q modulo q is a subset of $[0..q)$ such that all values in $[0..q)$ can be expressed as a difference of two elements in D_q modulo q. In other words:

$$[0..q) = \{i - j \bmod q \mid i, j \in D_q\} \ .$$

Example 5.14: $D_7 = \{1, 2, 4\}$

$$\begin{array}{ll} 1-1=0 & 1-4=-3\equiv 4 \pmod q \\ 2-1=1 & 2-4=-2\equiv 5 \pmod q \\ 4-2=2 & 1-2=-1\equiv 6 \pmod q \\ 4-1=3 & \end{array}$$

In general, we want the smallest possible difference cover for a given q.

- For any q, there exist a difference cover D_q of size $\mathcal{O}(\sqrt{q})$.
- The DC3 algorithm uses the simplest non-trivial difference cover $D_3 = \{1, 2\}$.

A **difference cover sample** is a set T_C of suffixes, where

$$C = \{i \in [0..n] \mid i \bmod q \in D_q\} \ .$$

Example 5.15: If $T = \texttt{banana\$}$ and $D_3 = \{1, 2\}$,
then $C = \{1, 2, 4, 5\}$ and $T_C = \{\texttt{anana\$}, \texttt{nana\$}, \texttt{na\$}, \texttt{a\$}\}$.

Once we have sorted the difference cover sample T_C, we can compare any two suffixes in $\mathcal{O}(q)$ time.

Example 5.16: $D_3 = \{1, 2\}$ and $C = \{1, 2, 4, 5, \ldots\}$

$T_0 = T[0]T_1$	$T_0 = T[0]T[1]T_2$	$T_0 = T[0]T_1$
$T_1 = T[1]T_2$	$T_2 = T[2]T[3]T_4$	$T_3 = T[3]T_4$

There is a tradeoff in choosing q, because we want to

- minimize comparison time $\mathcal{O}(q)$ and
- minimize sample size $|C| \leq \frac{|D_q|}{q}(n + 1) = \mathcal{O}\left(\frac{n}{\sqrt{q}}\right)$.

With DC3, it is enough that $|C| \leq \frac{2}{3}(n + 1)$.

Algorithm 5.17: DC3

Step 0: Choose C.

- Use difference cover $D_3 = \{1, 2\}$.
- For $k \in \{0, 1, 2\}$, define $C_k = \{i \in [0..n] \mid i \bmod 3 = k\}$.
- Let $C = C_1 \cup C_2$ and $\bar{C} = C_0$.

Example 5.18:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	y	a	b	b	a	d	a	b	b	a	d	o	$

$\bar{C} = C_0 = \{0, 3, 6, 9, 12\}$, $C_1 = \{1, 4, 7, 10\}$, $C_2 = \{2, 5, 8, 11\}$ and $C = \{1, 2, 4, 5, 7, 8, 10, 11\}$.

Step 1: Sort T_C.

- For $k \in \{1,2\}$, Construct the strings $R_k = (T_k^3, T_{k+3}^3, T_{k+6}^3, \ldots, T_{\max C_k}^3)$ whose characters are factors of length 3 in the original text, and let $R = R_1 R_2$.
- Replace each factor T_i^3 in R with a lexicographic name $N_i^3 \in [1..|R|]$. The names can be computed by sorting the factors with LSD radix sort in $\mathcal{O}(n)$ time. Let R' be the result appended with 0.
- Construct the inverse suffix array $SA_{R'}^{-1}$ of R'. This is done recursively unless all symbols in R' are unique, in which case $SA_{R'}^{-1} = R'$.
- From $SA_{R'}^{-1}$, we get lexicographic names for suffixes in T_C. For $i \in C$, let $N_i = SA_{R'}^{-1}[j]$, where j is the position of T_i^3 in R. For $i \in \bar{C}$, let $N_i = \bot$. Also let $N_{n+1} = N_{n+2} = 0$.

Example 5.19:

R	abb	ada	bba	do$	bba	dab	bad	o$	
R'	1	2	4	7	4	6	3	8	0
$SA_{R'}^{-1}$	1	2	5	7	4	6	3	8	0

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$T[i]$	y	a	b	b	a	d	a	b	b	a	d	o	$		
N_i	$\bot$	1	4	$\bot$	2	6	$\bot$	5	3	$\bot$	7	8	$\bot$	0	0

Step 2(a): Sort $T_{\bar{C}}$.

- For each $i \in \bar{C}$, we represent T_i with the pair $(T[i], N_{i+1})$. Then

$$T_i \leq T_j \iff (T[i], N_{i+1}) \leq (T[j], N_{j+1}) \ .$$

  Note that $N_{i+1} \neq \perp$ for all i.

- The pairs $(T[i], N_{i+1})$ are sorted by LSD radix sort in $\mathcal{O}(n)$ time.

Example 5.20:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	y	a	b	b	a	d	a	b	b	a	d	o	$
N_i	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$

$T_{12} < T_6 < T_9 < T_3 < T_0$ because $(\texttt{\$}, 0) < (\texttt{a}, 5) < (\texttt{a}, 7) < (\texttt{b}, 2) < (\texttt{y}, 1)$.

Step 2(b): Merge T_C and $T_{\bar{C}}$.

- Use comparison based merging algorithm needing $\mathcal{O}(n)$ comparisons.
- To compare $T_i \in T_C$ and $T_j \in T_{\bar{C}}$, we have two cases:

$$i \in C_1 : T_i \le T_j \iff (T[i], N_{i+1}) \le (T[j], N_{j+1})$$
$$i \in C_2 : T_i \le T_j \iff (T[i], T[i+1], N_{i+2}) \le (T[j], T[j+1], N_{j+2})$$

  Note that for all i, $N_{i+1} \neq \perp$ in the first case and $N_{i+2} \neq \perp$ in the second case.

Example 5.21:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	y	a	b	b	a	d	a	b	b	a	d	o	$
N_i	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$

$T_1 < T_6$ because $(\mathtt{a}, 4) < (\mathtt{a}, 5)$.
$T_3 < T_8$ because $(\mathtt{b}, \mathtt{a}, 6) < (\mathtt{b}, \mathtt{a}, 7)$.

Theorem 5.22: Algorithm DC3 constructs the suffix array of a string $T[0..n)$ in $\mathcal{O}(n)$ time plus the time needed to sort the characters of T.

There are many variants:

- DC3 is an optimal algorithm under several parallel and external memory computation models, too. There exists both parallel and external memory implementations of DC3.

- Using a larger value of q, we obtain more space efficient algorithms. For example, using $q = \log n$, the time complexity is $\mathcal{O}(n \log n)$ and the space needed in addition to the text and the suffix array is $\mathcal{O}(n/\sqrt{\log n})$.