Recursive Suffix Array Construction

Let us now describe 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 as follows:

  (a) Construct a reduced string R of length $|C|$, whose characters are order preserving names of text factors starting at the positions in C.

  (b) Construct the suffix array of R recursively.

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

Assume that

- $|C| \leq \alpha n$ for a constant $\alpha < 1$, and
- excluding the recursive call, all steps in the algorithm take linear time.

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

To make the scheme work, the set C must satisfy two nontrivial conditions:

1. There exists an appropriate reduced string R.
2. Given sorted T_C the suffix array of T is easy to construct.

Finding sets C that satisfy both conditions is difficult, but there are two different methods 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 4.15: $D_7 = \{1, 2, 4\}$

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

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 4.16: 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. To compare suffixes T_i and T_j:

- If $i \in C$ and $j \in C$, then we already know their order from T_C.
- Otherwise, find ℓ such that $i + \ell \in C$ and $j + \ell \in C$. There always exists such $\ell \in [0..q)$. Then compare:

$$T_i = T[i..i + \ell)T_{i+\ell}$$
$$T_j = T[j..j + \ell)T_{j+\ell}$$

  That is, compare first $T[i..i + \ell)$ to $T[j..j + \ell)$, and if they are the same, then $T_{i+\ell}$ to $T_{j+\ell}$ using the sorted T_C.

Example 4.17: $D_3 = \{1, 2\}$ and $C = \{1, 2, 4, 5, \dots\}$

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

Algorithm 4.18: 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 4.19:

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 $R_k = (T^3_k, T^3_{k+3}, T^3_{k+6}, \ldots, T^3_{\max C_k})$ whose characters are 3-factors of the text, and let $R = R_1 R_2$.
- Replace each factor T^3_i in R with an order preserving name $N^3_i \in [1..|R|]$. 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 $SA^{-1}_{R'}$ of R'. This is done recursively using DC3 unless all symbols in R' are unique, in which case $SA^{-1}_{R'} = R'$.
- From $SA^{-1}_{R'}$, we get order preserving names for suffixes in T_C.
  For $i \in C$, let $N_i = SA^{-1}_{R'}[j]$, where j is the position of T^3_i in R.
  For $i \in \bar{C}$, let $N_i = \bot$. Also let $N_{n+1} = N_{n+2} = 0$.

Example 4.20:

R	abb	ada	bba	do$	bba	dab	bad	o$	
R'	1	2	4	7	4	6	3	8	0
$SA^{-1}_{R'}$	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 $N_{i+1} \neq \bot$ for all $i \in \bar{C}$.

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

Example 4.21:

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	$\bot$	1	4	$\bot$	2	6	$\bot$	5	3	$\bot$	7	8	$\bot$

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

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:

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

  Note that none of the N-values is $\perp$.

Example 4.22:

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 4.23: 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 $\mathcal{O}(n/\sqrt{\log n})$.

Induced Sorting

Define three type of suffixes $-$, $+$ and $*$ as follows:

$$C^- = \{i \in [0..n) \mid T_i > T_{i+1}\}$$
$$C^+ = \{i \in [0..n) \mid T_i < T_{i+1}\}$$
$$C^* = \{i \in C^+ \mid i-1 \in C^-\}$$

Example 4.24:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$T[i]$	m	m	i	s	s	i	s	s	i	i	p	p	i	i	$
type of T_i	$-$	$-$	$*$	$-$	$-$	$*$	$-$	$-$	$*$	$+$	$-$	$-$	$-$	$-$	

For every $a \in \Sigma$ and $x \in \{-,+.*\}$ define

$$C_a = \{i \in [0..n] \mid T[i] = a\}$$
$$C_a^x = C_a \cap C^x$$

Then

$$C_a^- = \{i \in C_a \mid T_i < a^\infty\}$$
$$C_a^+ = \{i \in C_a \mid T_i > a^\infty\}$$

and thus, if $i \in C_a^-$ and $j \in C_a^+$, then $T_i < T_j$. Hence the suffix array is $nC_1C_2 \ldots C_{\sigma-1} = nC_1^-C_1^+C_2^-C_2^+ \ldots C_{\sigma-1}^-C_{\sigma-1}^+$.

The basic idea of induced sorting is to use information about the order of T_i to **induce** the order of the suffix $T_{i-1} = T[i-1]T_i$. The main steps are:

1. Sort the sets C_a^*, $a \in [1..\sigma)$.
2. Use C_a^*, $a \in [1..\sigma)$, to induce the order of the sets C_a^-, $a \in [1..\sigma)$.
3. Use C_a^-, $a \in [1..\sigma)$, to induce the order of the sets C_a^+, $a \in [1..\sigma)$.

The suffixes involved in the induction steps can be indentified using the following rules (proof is left as an exercise).

Lemma 4.25: For all $a \in [1..\sigma)$

(a) $i - 1 \in C_a^-$ iff $i > 0$ and $T[i-1] = a$ and one of the following holds

1. $i = n$
2. $i \in C^*$
3. $i \in C^-$ and $T[i-1] \geq T[i]$.

(b) $i - 1 \in C_a^+$ iff $i > 0$ and $T[i-1] = a$ and one of the following holds

1. $i \in C^-$ and $T[i-1] < T[i]$
2. $i \in C^+$ and $T[i-1] \leq T[i]$.

To induce C^- suffixes:

1. Set C_a^- empty for all $a \in [1..\sigma)$.
2. For all suffixes T_i such that $i-1 \in C^-$ **in lexicographical order**, append $i-1$ into $C^-_{T[i-1]}$.

By Lemma 4.25(a), Step 2 can be done by checking the relevant conditions for all $i \in nC_1^- C_1^* C_2^- C_2^* \ldots$.

Algorithm 4.26: InduceMinusSuffixes
Input: Lexicographically sorted lists C_a^*, $a \in \Sigma$
Output: Lexicographically sorted lists C_a^-, $a \in \Sigma$

(1) **for** $a \in \Sigma$ **do** $C_a^- \leftarrow \emptyset$
(2) *pushback*$(n-1, C^-_{T[n-1]})$
(3) **for** $a \leftarrow 1$ **to** $\sigma - 1$ **do**
(4)   **for** $i \in C_a^-$ **do** // include elements added during the loop
(5)   **if** $i > 0$ **and** $T[i-1] \geq a$ **then** *pushback*$(i-1, C^-_{T[i-1]})$
(6)   **for** $i \in C_a^*$ **do** *pushback*$(i-1, C^-_{T[i-1]})$

Note that since $T_{i-1} > T_i$ by definition of C^-, we always have i inserted before $i-1$.

Inducing $+$-type suffixes goes similarly but in reverse order so that again i is always inserted before $i-1$:

1. Set C_a^+ empty for all $a \in [1..\sigma)$.

2. For all suffixes T_i such that $i-1 \in C^+$ in **descending** lexicographical order, append $i-1$ into $C^+_{T[i-1]}$.

Algorithm 4.27: InducePlusSuffixes
Input: Lexicographically sorted lists C_a^-, $a \in \Sigma$
Output: Lexicographically sorted lists C_a^+, $a \in \Sigma$

for $a \in \Sigma$ do $C_a^+ \leftarrow \emptyset$
for $a \leftarrow \sigma - 1$ downto 1 do
    for $i \in C_a^+$ in reverse order do // include elements added during loop
        if $i > 0$ and $T[i-1] \le a$ then *pushfront*$(i-1, C^+_{T[i-1]})$
    for $i \in C_a^-$ in reverse order do
        if $i > 0$ and $T[i-1] < a$ then *pushfront*$(i-1, C^+_{T[i-1]})$

We still need to explain how to sort the $*$-type suffixes. Define

$$F[i] = \min\{k \in [i+1..n] \mid k \in C^* \text{ or } k = n\}$$
$$S_i = T[i..F[i]]$$
$$S_i' = S_i\sigma$$

where σ is a special symbol larger than any other symbol.

Lemma 4.28: For any $i, j \in [0..n)$, $T_i < T_j$ iff $S_i' < S_j'$ or $S_i' = S_j'$ and $T_{F[i]} < T_{F[j]}$.

Proof. The claim is trivially true except in the case that S_j is a proper prefix of S_i (or vice versa). In that case, $S_i > S_j$ but $S_i' < S_j'$ and thus $T_i < T_j$ by the claim. We will show that this is correct.

Let $\ell = F[j]$ and $k = i + \ell - j$. Then

- $\ell \in C^*$ and thus $\ell - 1 \in C^-$. By Lemma 4.25(b), $T[\ell - 1] > T[\ell]$.
- $T[k-1..k] = T[\ell-1..\ell]$ and thus $T[k-1] > T[k]$. If we had $k \in C^+$, we would have $k \in C^*$. Since this is not the case, we must have $k \in C^-$.
- Let $a = T[\ell]$. Since $\ell \in C_a^+$ and $k \in C_a^-$, we must have $T_k < a^\infty < T_\ell$.
- Since $T[i..k) = T[j..\ell)$ and $T_k < T_\ell$, we have $T_i < T_j$.

$\square$

Algorithm 4.29: SAIS

Step 0: Choose C.

- Compute the types of suffixes. This can be done in $\mathcal{O}(n)$ time based on Lemma 4.25.
- Set $C = \cup_{a\in[1..\sigma)} C^*_a \cup \{n\}$. Note that $|C| \leq n/2$, since for all $i \in C$, $i-1 \in C^- \subseteq \bar{C}$.

Example 4.30:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$T[i]$	m	m	i	s	s	i	s	s	i	i	p	p	i	i	$
type of T_i	−	−	∗	−	−	∗	−	−	∗	+	−	−	−	−	

$C^*_{\mathtt{i}} = \{2, 5, 8\}$, $C^*_{\mathtt{m}} = C^*_{\mathtt{p}} = C^*_{\mathtt{s}} = \emptyset$, $C = \{2, 5, 8, 14\}$.

Step 1: Sort T_C.

- Sort the strings S'_i, $i \in C^*$. Since the total length of the strings S'_i is $\mathcal{O}(n)$, the sorting can be done in $\mathcal{O}(n)$ time using LSD radix sort.
- Assign order preserving names $N_i \in [1..|C|-1]$ to the string S'_i so that $N_i \leq N_j$ iff $S'_i \leq S'_j$.
- Construct the sequence $R = N_{i_1} N_{i_2} \ldots N_k 0$, where $i_1 < i_3 < \cdots < i_k$ are the *-type positions.
- Construct the suffix array SA_R of R. This is done recursively unless all symbols in R are unique, in which case a simple counting sort is sufficient.
- The order of the suffixes of R corresponds to the order of $*$-type suffixes of T. Thus we can construct the lexicographically ordered lists C^*_a, $a \in [1..\sigma)$.

Example 4.31:

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

$R = [\mathtt{issi}\sigma][\mathtt{issi}\sigma][\mathtt{iippii\$}\sigma]\mathtt{\$} = 2210$, $SA_R = (3,2,1,0)$, $C^*_{\mathtt{i}} = (8,5,2)$

Step 2: Sort $T_{[0..n]}$.

- Run InduceMinusSuffixes to construct the sorted lists C_a^-, $a \in [1..\sigma)$.
- Run InducePlusSuffixes to construct the sorted lists C_a^+, $a \in [1..\sigma)$.
- The suffix array is $SA = nC_1^- C_1^+ C_2^- C_2^+ \dots C_{\sigma-1}^- C_{\sigma-1}^+$.

Example 4.32:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$T[i]$	m	m	i	s	s	i	s	s	i	i	p	p	i	i	$
type of T_i	−	−	∗	−	−	∗	−	−	∗	+	−	−	−	−	

$n = 14 \quad \Rightarrow \quad C_{\mathtt{i}}^- = (13, 12)$

$C_{\mathtt{i}}^- C_{\mathtt{i}}^* = (13, 12, 8, 5, 2) \quad \Rightarrow \quad C_{\mathtt{m}}^- = (1, 0),\ C_{\mathtt{p}}^- = (11, 10),\ C_{\mathtt{s}}^- = (7, 4, 6, 3)$

$\Rightarrow \quad C_{\mathtt{i}}^+ = (8, 9, 5, 2)$

$\Rightarrow \quad SA = C_{\$} C_{\mathtt{i}}^- C_{\mathtt{i}}^+ C_{\mathtt{m}}^- C_{\mathtt{p}}^- C_{\mathtt{s}}^- = (14, 13, 12, 8, 9, 5, 2, 1, 0, 11, 10, 7, 4, 6, 3)$

Theorem 4.33: Algorithm SAIS 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.

- In Step 1, to sort the strings S'_i, $i \in C^*$, SAIS does not actually use LSD radix sort but the following procedure:

  1. Construct the sets C^*_a, $a \in [1..\sigma)$ **in arbitrary order**.
  2. Run InduceMinusSuffixes to construct the lists C^-_a, $a \in [1..\sigma)$.
  3. Run InducePlusSuffixes to construct the lists C^-_a, $a \in [1..\sigma)$.
  4. Remove non-*-type positions from $C^+_1 C^+_2 \ldots C^+_{\sigma-1}$.

  With this change, most of the work is done in the induction procedures. This is very fast in practice, because all the lists C^x_a are accessed **sequentially** during the procedures.

- The currently fastest suffix sorting implementation in practice is probably divsufsort by Yuta Mori. It sorts the *-type suffixes non-recursively in $\mathcal{O}(n \log n)$ time and then continues as SAIS.

Summary: Suffix Trees and Arrays

The most important data structures for string processing:

- Designed for indexed exact string matching.
- Used in efficient solutions to a huge variety of different problems.

Construction algorithms are among the most important algorithms for string processing:

- Linear time for constant and integer alphabet.

Often augmented with additional data structures:

- suffix links, LCA preprocessing
- LCP array, RMQ preprocessing, BWT, ...