2.1 Linear time suffix array construction

This exposition has been developed by David Weese. It is based on the following sources, which are all recommended reading:

1. J. Kärkkäinen, P. Sanders (2003) *Simple linear work suffix array construction*, In Proc. ICALP '03. LNCS 2719, pp. 943–955
2. J. Kärkkäinen, P. Sanders, S. Burkhardt (2006) *Linear work suffix array construction*, Journal of the ACM, 53(6): 918–936

2.2 Definitions

We consider a string T of length n. For $i, j \in \mathbb{N}_0$ we define:

- $[i..j] := \{i, i+1, \ldots, j\}$
- $[i..j) := [i..j-1]$
- $T[i]$ is the i-th character of T.
- $T[i..j] := T[i]T[i+1] \ldots T[j]$ is the substring from the i-th to the j-th character
- We start counting from **0**, i. e. $T = T[0..n-1]$
- $|T|$ denotes the string length, i. e. $|T| = n$
- The concatenation of strings X, Y is denoted as $X \cdot Y$, e. g. $T = T[0..i-1] \cdot T[i..n-1]$ for $i \in [1..n)$

2.3 Lexicographical naming

Definition 1. Given a set of strings $\mathcal{S}$. A map $\phi : \mathcal{S} \to [0..|\mathcal{S}|)$ is called *lexicographical naming* if for every $X, Y \in \mathcal{S}$ holds: $X <_{\text{lex}} Y \Leftrightarrow \phi(X) < \phi(Y)$. We call $\phi(X)$ the *name* or *rank* of X.

The skew algorithm uses the following lemma to reduce the lex. relation of concatenated strings to the relation of the concatenation of names.

Lemma 2. *Given a set $\mathcal{S} \subseteq \Sigma^t$ of strings having length t and a lex. naming ϕ for $\mathcal{S}$. Let $X_1, \ldots, X_k \in \mathcal{S}$ and $Y_1, \ldots, Y_l \in \mathcal{S}$ be strings from $\mathcal{S}$. The lexicographical relation of the concatenated strings $X_1 \cdot X_2 \cdots X_k$ and $Y_1 \cdot Y_2 \cdots Y_l$ equals the lex. relation of the strings of names:*

$$\begin{array}{rrcl} & X_1 \cdot X_2 \cdot \cdots \cdot X_k & <_{lex} & Y_1 \cdot Y_2 \cdots Y_l \\ \Leftrightarrow & \phi(X_1)\phi(X_2)\ldots\phi(X_k) & <_{lex} & \phi(Y_1)\phi(Y_2)\ldots\phi(Y_l) \end{array}$$

2.4 Outline of the skew algorithm

1. Construct the suffix array A^{12} of the suffixes starting at positions $i \not\equiv 0 \pmod 3$. This is done by a recursive call of the skew algorithm for a string of two thirds the length.
2. Construct the suffix array A^0 of the remaining suffixes using the result of the first step.
3. Merge the two suffix arrays into one.

2.5 *Step 1:* Construct the suffix array A^{12}

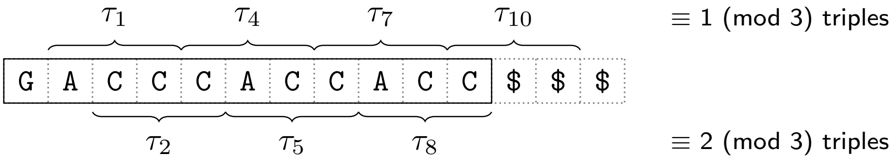
We consider a text T of length n and want to create the suffix array A^{12} for suffixes $T[i..n-1]$ where $0 < i < n$ and $i \not\equiv 0 \pmod 3$.

In order to call the suffix array algorithm recursively we construct a new text T' whose suffix array can be used to derive A^{12}. This is done as follows:

1. (a) Lexicographically name all triples $T[i..i+2]$
   (b) Construct a text T' of triple names
   (c) Construct suffix array A' of T' (recursively)
   (d) Transform A' into A^{12}

2.6 *Step 1a:* Lexicographically name triples

A *triple* is a substring of length 3. In the following we only consider triples $T[i..i+2]$ with $i \not\equiv 0 \pmod 3$. Let \$ be a character that is not contained in T and less than every other character. We append \$\$\$ to T to obtain well-defined triples even for $i \in [n-2..n]$



We lexicographically sort the triples using 3 passes of radix sort. Hereafter we assign τ_i the lex. rank of the triple $T[i..i+2]$. The τ_i are now *lexicographical names* of the triples.

Example (T = GACCCACCACC): Initialize list of triple start positions with $\left\langle i \mid i \in \left[1..n+(n_0-n_1)\right) \wedge i \not\equiv 0 \pmod 3 \right\rangle = \langle 1, 2, 4, 5, 7, 8, 10 \rangle$. Sort list with radix sort:

i	$T[i..i+2]$
1	ACC
2	CCC
4	CAC
5	ACC
7	CAC
8	ACC
10	C\$\$

$\xrightarrow{\text{radix pass}}$

i	$T[i..i+2]$
10	C**\$\$**
1	A**CC**
2	C**CC**
4	C**AC**
5	A**CC**
7	C**AC**
8	A**CC**

$\xrightarrow{\text{radix pass}}$

i	$T[i..i+2]$
10	**C\$\$**
4	**CAC**
7	**CAC**
1	**ACC**
2	**CCC**
5	**ACC**
8	**ACC**

$\xrightarrow{\text{radix pass}}$

i	$T[i..i+2]$	τ_i
1	**ACC**	0
5	**ACC**	0
8	**ACC**	0
10	**C\$\$**	1
4	**CAC**	2
7	**CAC**	2
2	**CCC**	3

2.7 *Step 1b:* Construct T'

$T' = t_1 t_2$ is the concatenation of strings t_1 and t_2 of triple names with

$$\begin{array}{rcl} t_1 & = & \tau_1\tau_4\ldots\tau_{1+3n_0} \\ t_2 & = & \tau_2\tau_5\ldots\tau_{2+3n_2} \end{array} \qquad \text{with} \qquad n_j = \left\lceil \frac{n-j}{3} \right\rceil$$

n_j for $j \in \{0, 1, 2\}$ is the number of triples starting at positions $i \equiv j \pmod 3$ that overlap with the first n text characters.

The last triple of t_1 and t_2 possibly ends with \$. To ensure that t_1 always ends with a separating \$, we in case $n \equiv 1 \pmod 3 \Leftrightarrow n_0 - n_1 = 1$ include the extra triple \$\$\$ into the set of triples (in Step 1a) and append its name to t_1 . Therefore t_1 contains $n_1 + (n_0 - n_1) = n_0$ triples names.

Now, there is a one-to-one correspondence between suffixes of T' and the (possibly empty) suffixes $T[i..n-1]$ with $i \not\equiv 0 \pmod 3$.

Example (T = GACCCACCACC): Construct $T' = \left\langle \tau_{1+3i} \mid i \in [0..n_0) \right\rangle \cdot \left\langle \tau_{2+3i} \mid i \in [0..n_2) \right\rangle$

$$\begin{array}{rcl} n & = & 11 \\ n_0 & = & \left\lceil \frac{11}{3} \right\rceil \quad = \quad 4 \\ n_2 & = & \left\lceil \frac{11-2}{3} \right\rceil \quad = \quad 3 \end{array}$$

$$\begin{array}{rcccccccc} T' & = & \tau_1 & \tau_4 & \tau_7 & \tau_{10} & \tau_2 & \tau_5 & \tau_8 \\ & = & 0 & 2 & 2 & 1 & 3 & 0 & 0 \\ & \widehat{=} & \mathtt{ACC} & \mathtt{CAC} & \mathtt{CAC} & \mathtt{C\$\$} & \mathtt{CCC} & \mathtt{ACC} & \mathtt{ACC} \end{array}$$

2.8 *Step 1c:* Construct the suffix array A' of T'

T' is a string of length $\left\lceil \frac{2n-1}{3} \right\rceil$ over the alphabet $[0..|T'|)$. We recursively use the skew algorithm to construct the suffix array A' of T'.

If the names τ_i are unique amongst the triples, A' can be directly be derived from T' without recursion (Exercise).

Example (T = GACCCACCACC):

$$T' = 0\ 2\ 2\ 1\ 3\ 0\ 0$$

$$\begin{array}{rclclcl} A'[0] & = & 6 & \widehat{=} & 0 & \widehat{=} & \mathtt{ACC} \\ A'[1] & = & 5 & \widehat{=} & 00 & \widehat{=} & \mathtt{ACCACC} \\ A'[2] & = & 0 & \widehat{=} & 0221300 & \widehat{=} & \mathtt{ACCCACCACC\$\$...} \\ A'[3] & = & 3 & \widehat{=} & 1300 & \widehat{=} & \mathtt{C\$\$...} \\ A'[4] & = & 2 & \widehat{=} & 21300 & \widehat{=} & \mathtt{CACC\$\$...} \\ A'[5] & = & 1 & \widehat{=} & 221300 & \widehat{=} & \mathtt{CACCACC\$\$...} \\ A'[6] & = & 4 & \widehat{=} & 300 & \widehat{=} & \mathtt{CCCACCACC} \end{array}$$

2.9 *Step 1d:* Transform A' into A^{12}

Suffixes starting at j in t_2 start at $i = j+n_0$ in T' and one-to-one correspond to suffixes starting at $2+3j = 2+3(i-n_0)$ in T. Hence they are in correct lex. order.

Suffixes starting at i in t_1 one-to-one correspond to suffixes starting at $1 + 3i$ in T. The t_2-tail has no influence on their order due to the unique triple at the end of t_1.

Transform A' into A^{12} such that:

$$A^{12}[i] = \begin{cases} 1 + 3A'[i] & \text{if } A'[i] < n_0 \\ 2 + 3(A'[i] - n_0) & \text{else} \end{cases}$$

Example (T = GACCCACCACC):

$$\begin{array}{rclcrcl} A'[0] & = & 6 & \longrightarrow & A^{12}[0] & = & 8 \\ A'[1] & = & 5 & \longrightarrow & A^{12}[1] & = & 5 \\ A'[2] & = & 0 & \longrightarrow & A^{12}[2] & = & 1 \\ A'[3] & = & 3 & \longrightarrow & A^{12}[3] & = & 10 \\ A'[4] & = & 2 & \longrightarrow & A^{12}[4] & = & 7 \\ A'[5] & = & 1 & \longrightarrow & A^{12}[5] & = & 4 \\ A'[6] & = & 4 & \longrightarrow & A^{12}[6] & = & 2 \end{array}$$

2.10 *Step 2:* Derive A^0 from A^{12}

Extract suffixes T_i with $i \equiv 1 \pmod 3$ from A^{12} and store $i-1$ in A^0 in the same order. Use a radix pass to stably sort A^0 by the first suffix character.

This gives the correct lexicographical order as for $i < j$ either

$$\begin{array}{lcll} T\left[A^0[i]\right] & < & T\left[A^0[j]\right] & \text{or} \\ T\left[A^0[i]\right] & = & T\left[A^0[j]\right] & \wedge \quad T\left[A^0[i]+1..n-1\right] \ <_{\text{lex}} \ T\left[A^0[j]+1..n-1\right] \ \text{ holds.} \end{array}$$

Example (T = `GACCCACCACC`):

$$\begin{array}{rcccccccc} A^{12} & = & 8 & 5 & 1 & 10 & 7 & 4 & 2 \\ A^{0} & = & & & 0 & 9 & 6 & 3 & \end{array}$$

$$\begin{array}{rclcl} A^0[0] & = & 0 & \mathrel{\widehat{=}} & \texttt{GACCCACCACC} \\ A^0[1] & = & 9 & \mathrel{\widehat{=}} & \texttt{CC} \\ A^0[2] & = & 6 & \mathrel{\widehat{=}} & \texttt{CCACC} \\ A^0[3] & = & 3 & \mathrel{\widehat{=}} & \texttt{CCACCACC} \end{array} \quad \xrightarrow{\text{radix pass}} \quad \begin{array}{rclcl} A^0[0] & = & 9 & \mathrel{\widehat{=}} & \texttt{CC} \\ A^0[1] & = & 6 & \mathrel{\widehat{=}} & \texttt{CCACC} \\ A^0[2] & = & 3 & \mathrel{\widehat{=}} & \texttt{CCACCACC} \\ A^0[3] & = & 0 & \mathrel{\widehat{=}} & \texttt{GACCCACCACC} \end{array}$$

2.11 *Step 3:* Merge A^{12} and A^0 into suffix array A

The two sorted suffix arrays are merged by scanning them simultaneously and comparing the suffixes from A^0 and A^{12}. If $n \equiv 1 \pmod 3$, the first suffix of A^{12} must be skipped.

To determine the lex. rank of a suffix in A^{12} we construct the inverse R^{12} of A^{12} such that $R^{12}[A^{12}[i]] = i$ and $R^{12}[n] = 0$. Two suffixes $i \in A^0$ and $j \in A^{12}$ can be compared using:

Case 1: $i \equiv 0 \pmod 3$ and $j \equiv 1 \pmod 3$

$$T[i..n-1] <_{\text{lex}} T[j..n-1] \quad \Leftrightarrow \quad \begin{array}{l} \big(T[i] < T[j]\big) \quad \vee \\ \big(T[i] = T[j] \ \wedge \ R^{12}[i+1] < R^{12}[j+1]\big) \end{array}$$

The rank comparison is possible as $i+1 \equiv 1 \pmod 3$ and $j+1 \equiv 2 \pmod 3$.

Case 2: $i \equiv 0 \pmod 3$ and $j \equiv 2 \pmod 3$

$$T[i..n-1] <_{\text{lex}} T[j..n-1] \quad \Leftrightarrow \quad \begin{array}{l} \big(T[i..i+1] <_{\text{lex}} T[j..j+1]\big) \quad \vee \\ \big(T[i..i+1] =_{\text{lex}} T[j..j+1] \ \wedge \ R^{12}[i+2] < R^{12}[j+2]\big) \end{array}$$

The rank comparison is possible as $i+2 \equiv 2 \pmod 3$ and $j+2 \equiv 1 \pmod 3$.

Example (T = `GACCCACCACC`):

	0	1	2	3	4	5	6	7	8	9	10	11	12
T	G	A	C	C	C	A	C	C	A	C	C	$	$
R^{12}		3	7		6	2		5	1		4	0	

$$\begin{array}{rcccccccc} & & \downarrow & & & & & & \\ A^{12} & = & 8 & 5 & 1 & 10 & 7 & 4 & 2 \\ A^{0} & = & 9 & 6 & 3 & 0 & & & \\ & & \uparrow & & & & & & \end{array}$$

If $n \equiv 1 \pmod 3$, skip the first element of A^{12} (this is not the case).

Compare T_8 with T_9:
$T[8..9] = \texttt{AC} <_{\text{lex}} \texttt{CC} = T[9..10] \quad \Rightarrow \quad A[0] = 8$

$$A = 8$$

$$\begin{array}{rcccccccc} & & & \downarrow & & & & & \\ A^{12} & = & 8 & 5 & 1 & 10 & 7 & 4 & 2 \\ A^{0} & = & 9 & 6 & 3 & 0 & & & \\ & & \uparrow & & & & & & \end{array}$$

Compare T_5 with T_9:
$T[5..6] = \texttt{AC} <_{\text{lex}} \texttt{CC} = T[9..10] \quad \Rightarrow \quad A[1] = 5$

$$A = 8 \quad 5$$

$$\begin{array}{rcccccccc} & & & & \downarrow & & & & \\ A^{12} & = & 8 & 5 & 1 & 10 & 7 & 4 & 2 \\ A^{0} & = & 9 & 6 & 3 & 0 & & & \\ & & \uparrow & & & & & & \end{array}$$

Compare T_1 with T_9:
$T[1] = \texttt{A} < \texttt{C} = T[9] \quad \Rightarrow \quad A[2] = 1$

$$A = 8 \quad 5 \quad 1$$

$$\begin{array}{rcccccccc} & & & & & \downarrow & & & \\ A^{12} & = & 8 & 5 & 1 & 10 & 7 & 4 & 2 \\ A^{0} & = & 9 & 6 & 3 & 0 & & & \\ & & \uparrow & & & & & & \end{array}$$

Compare T_{10} with T_9:

$$\begin{array}{rcccccccl} T[10] & = & \texttt{C} & = & \texttt{C} & = & T[9] & \wedge & \\ R^{12}[11] & = & 0 & < & 4 & = & R^{12}[10] & \Rightarrow & A[3] = 10 \end{array}$$

$$A = 8 \quad 5 \quad 1 \quad 10$$

$$\begin{array}{rcccccccc} & & & & & & \downarrow & & \\ A^{12} & = & 8 & 5 & 1 & 10 & 7 & 4 & 2 \\ A^{0} & = & 9 & 6 & 3 & 0 & & & \\ & & \uparrow & & & & & & \end{array}$$

Compare T_7 with T_9:

$$\begin{array}{rcccccccl} T[7] & = & \texttt{C} & = & \texttt{C} & = & T[9] & \wedge & \\ R^{12}[8] & = & 1 & < & 4 & = & R^{12}[10] & \Rightarrow & A[4] = 7 \end{array}$$

$$A = 8 \quad 5 \quad 1 \quad 10 \quad 7$$

$$\begin{array}{rcccccccc} & & & & & & & \downarrow & \\ A^{12} & = & 8 & 5 & 1 & 10 & 7 & 4 & 2 \\ A^{0} & = & 9 & 6 & 3 & 0 & & & \\ & & \uparrow & & & & & & \end{array}$$

Compare T_4 with T_9:

$$\begin{array}{rcccccccl} T[4] & = & \texttt{C} & = & \texttt{C} & = & T[9] & \wedge & \\ R^{12}[5] & = & 2 & < & 4 & = & R^{12}[10] & \Rightarrow & A[5] = 4 \end{array}$$

$$A = 8 \quad 5 \quad 1 \quad 10 \quad 7 \quad 4$$

$$\begin{array}{rcccccccc} & & & & & & & & \downarrow \\ A^{12} & = & 8 & 5 & 1 & 10 & 7 & 4 & 2 \\ A^{0} & = & 9 & 6 & 3 & 0 & & & \\ & & \uparrow & & & & & & \end{array}$$

Compare T_2 with T_9:

$$\begin{array}{rcccccccl} T[2..3] & = & \texttt{CC} & =_{\text{lex}} & \texttt{CC} & = & T[9..10] & \wedge & \\ R^{12}[4] & = & 6 & > & 0 & = & R^{12}[11] & \Rightarrow & A[6] = 9 \end{array}$$

$$A = 8 \quad 5 \quad 1 \quad 10 \quad 7 \quad 4 \quad 9$$

$$\begin{array}{rcccccccc} & & & & & & & & \downarrow \\ A^{12} & = & 8 & 5 & 1 & 10 & 7 & 4 & 2 \\ A^{0} & = & 9 & 6 & 3 & 0 & & & \\ & & & \uparrow & & & & & \end{array}$$

Compare T_2 with T_6:

$$\begin{array}{rcccccccl} T[2..3] & = & \mathtt{CC} & =_{\text{lex}} & \mathtt{CC} & = & T[6..7] & \wedge & \\ R^{12}[4] & = & 6 & > & 1 & = & R^{12}[8] & \Rightarrow & A[7] = 6 \end{array}$$

$$A \;=\; 8 \quad 5 \quad 1 \quad 10 \quad 7 \quad 4 \quad 9 \quad 6$$

$$\begin{array}{rcccccccc} & & & & & & & & \downarrow \\ A^{12} & = & 8 & 5 & 1 & 10 & 7 & 4 & 2 \\ A^{0} & = & 9 & 6 & 3 & 0 & & & \\ & & & & \uparrow & & & & \end{array}$$

Compare T_2 with T_3:

$$\begin{array}{rcccccccl} T[2..3] & = & \mathtt{CC} & =_{\text{lex}} & \mathtt{CC} & = & T[3..4] & \wedge & \\ R^{12}[4] & = & 6 & > & 2 & = & R^{12}[5] & \Rightarrow & A[8] = 3 \end{array}$$

$$A \;=\; 8 \quad 5 \quad 1 \quad 10 \quad 7 \quad 4 \quad 9 \quad 6 \quad 3$$

$$\begin{array}{rcccccccc} & & & & & & & & \downarrow \\ A^{12} & = & 8 & 5 & 1 & 10 & 7 & 4 & 2 \\ A^{0} & = & 9 & 6 & 3 & 0 & & & \\ & & & & & \uparrow & & & \end{array}$$

Compare T_2 with T_0:

$T[2..3] = \mathtt{CC} <_{\text{lex}} \mathtt{GA} = T[0..1] \quad \Rightarrow \quad A[9] = 2$

$$A \;=\; 8 \quad 5 \quad 1 \quad 10 \quad 7 \quad 4 \quad 9 \quad 6 \quad 3 \quad 2$$

$$\begin{array}{rccccccccc} & & & & & & & & & \downarrow \\ A^{12} & = & 8 & 5 & 1 & 10 & 7 & 4 & 2 & \\ A^{0} & = & 9 & 6 & 3 & 0 & & & & \\ & & & & & \uparrow & & & & \end{array}$$

All characters of A^{12} were read. Fill up A with the remainder of A^0.

$$A \;=\; 8 \quad 5 \quad 1 \quad 10 \quad 7 \quad 4 \quad 9 \quad 6 \quad 3 \quad 2 \quad 0$$

Done. The resulting suffix array is:

$$\begin{array}{rcrcl} A[0] & = & 8 & \widehat{=} & \mathtt{ACC} \\ A[1] & = & 5 & \widehat{=} & \mathtt{ACCACC} \\ A[2] & = & 1 & \widehat{=} & \mathtt{ACCCACCACC} \\ A[3] & = & 10 & \widehat{=} & \mathtt{C} \\ A[4] & = & 7 & \widehat{=} & \mathtt{CACC} \\ A[5] & = & 4 & \widehat{=} & \mathtt{CACCACC} \\ A[6] & = & 9 & \widehat{=} & \mathtt{CC} \\ A[7] & = & 6 & \widehat{=} & \mathtt{CCACC} \\ A[8] & = & 3 & \widehat{=} & \mathtt{CCACCACC} \\ A[9] & = & 2 & \widehat{=} & \mathtt{CCCACCACC} \\ A[10] & = & 0 & \widehat{=} & \mathtt{GACCCACCACC} \end{array}$$

2.12 Linear running time

Assuming that $|\Sigma| = O(n)$, the running time $\mathcal{T}(n)$ of the whole skew-algorithm is the sum of:

- A recursive part which takes $\mathcal{T}(\frac{2n}{3})$ time.
- A non-recursive part which takes $O(n)$ time.

Thus it holds: $\mathcal{T}(n) = \mathcal{T}(\frac{2n}{3}) + O(n)$ and $\mathcal{T}(n) = O(1)$ for $n \leq 3$.

Lemma 3. *The running time of the skew algorithm is* $\mathcal{T}(n) = O(n)$.

Proof: Exercise.

2.13 Difference Covers

The key idea of the skew algorithm is to

1. recursively sort a subset $\mathcal{I} \subset \mathcal{R}$ of congruence class ring $\mathcal{R}$
2. deduce the sorting of the remaining classes $\mathcal{R} \setminus \mathcal{I}$.
3. merge $\mathcal{I}$ and $\mathcal{R} \setminus \mathcal{I}$

In the original skew algorithm holds $\mathcal{R} = \mathbb{Z}_3 = \{3\mathbb{Z}, 1+3\mathbb{Z}, 2+3\mathbb{Z}\}$ and $\mathcal{I} = \{\mathbf{1}+3\mathbb{Z}, \mathbf{2}+3\mathbb{Z}\}$. Step 3 was feasible because for every $x \in \mathcal{I}$ and $y \in \mathcal{R} \setminus \mathcal{I}$ there was a $\Delta \in \mathbb{N}$ such that $(x + \Delta) \in \mathcal{I}$ and $(y + \Delta) \in \mathcal{I}$.

The recursion depth of the skew algorithm heavily depends on $\lambda = \frac{|\mathcal{I}|}{|\mathcal{R}|}$ the factor the text length decreases with. Is it possible to find $\mathcal{I}$ and $\mathcal{R}$ yielding a smaller λ and such that step 2 and 3 are still feasible?

Definition 4. For a set of congruence classes $\mathcal{R} = \{m\mathbb{Z}, 1+m\mathbb{Z}, \ldots, (m-1)+m\mathbb{Z}\}$ we call $\mathcal{I}$ to be *difference cover* if for any $z \in \mathcal{R}$ there exist $a, b \in \mathcal{I}$ such that $a - b = z$.

Lemma 5. *Step 3 of the skew algorithm is feasible for any* m*, if* $\mathcal{I}$ *is a difference cover of* $\mathcal{R}$*.*

Proof: For any $x, y \in \mathcal{R}$ there exist $a, b \in \mathcal{I}$ such that $a - b = z$ with $z = x - y$. For $\Delta := a - x$ holds

$$(x + \Delta) = x + (a - x) = a \quad \Rightarrow \quad (x + \Delta) \in \mathcal{I}$$

and

$$(y + \Delta) = y + (a - x) = a - (x - y) = a - z = b \quad \Rightarrow \quad (y + \Delta) \in \mathcal{I} \quad .$$

By combinatorics the size of a set $\mathcal{R}$ that is covered by $\mathcal{I}$ is limited to:

$$|\mathcal{R}| \leq 2 \cdot \binom{|\mathcal{I}|}{2} + 1 = |\mathcal{I}|^2 - |\mathcal{I}| + 1$$

We call $\mathcal{I}$ a *perfect difference cover* if $|\mathcal{R}| = |\mathcal{I}|^2 - |\mathcal{I}| + 1$ holds. The following table shows perfect difference covers in bold:

$\lvert\mathcal{I}\rvert$	$\mathcal{R}$	minimal difference cover	λ
2	$\mathbb{Z}_3$	$\{\mathbf{1}, \mathbf{2}\}$	0,6666...
3	$\mathbb{Z}_7$	$\{\mathbf{1}, \mathbf{2}, \mathbf{4}\}$	0,4285...
4	$\mathbb{Z}_{13}$	$\{\mathbf{1}, \mathbf{2}, \mathbf{4}, \mathbf{10}\}$	0,3076...
5	$\mathbb{Z}_{21}$	$\{\mathbf{1}, \mathbf{2}, \mathbf{7}, \mathbf{9}, \mathbf{19}\}$	0,2380...
6	$\mathbb{Z}_{31}$	$\{\mathbf{1}, \mathbf{2}, \mathbf{4}, \mathbf{9}, \mathbf{13}, \mathbf{19}\}$	0,1935...
7	$\mathbb{Z}_{39}$	$\{1, 2, 17, 21, 23, 28, 31\}$	0,1794...
8	$\mathbb{Z}_{57}$	$\{\mathbf{1}, \mathbf{2}, \mathbf{10}, \mathbf{12}, \mathbf{15}, \mathbf{36}, \mathbf{40}, \mathbf{52}\}$	0,1403...
9	$\mathbb{Z}_{73}$	$\{\mathbf{1}, \mathbf{2}, \mathbf{4}, \mathbf{8}, \mathbf{16}, \mathbf{32}, \mathbf{37}, \mathbf{55}, \mathbf{64}\}$	0,1232...
10	$\mathbb{Z}_{91}$	$\{\mathbf{1}, \mathbf{2}, \mathbf{8}, \mathbf{17}, \mathbf{28}, \mathbf{57}, \mathbf{61}, \mathbf{69}, \mathbf{71}, \mathbf{74}\}$	0,1098...
11	$\mathbb{Z}_{95}$	$\{1, 2, 6, 9, 19, 21, 30, 32, 46, 62, 68\}$	0,1157...
12	$\mathbb{Z}_{133}$	$\{\mathbf{1}, \mathbf{2}, \mathbf{33}, \mathbf{43}, \mathbf{45}, \mathbf{49}, \mathbf{52}, \mathbf{60}, \mathbf{73}, \mathbf{78}, \mathbf{98}, \mathbf{112}\}$	0,0902...

A next greater perfect difference cover is $\mathcal{I} = \{\mathbf{1} + 7\mathbb{Z}, \mathbf{2} + 7\mathbb{Z}, \mathbf{4} + 7\mathbb{Z}\}$ for $\mathcal{R} = \mathbb{Z}_7 = \{7\mathbb{Z}, 1 + 7\mathbb{Z}, \ldots, 6 + 7\mathbb{Z}\}$. It can be used with the following modifications to the original skew algorithm saving $\approx 20\%$ of running time:

1. Recursively sort the suffixes starting at $i \equiv 1, 2, 4 \pmod 7$.
2. Deduce the sorting of the remaining classes: $4 \rightarrow \mathbf{3}$ and $1 \rightarrow \mathbf{0} \rightarrow \mathbf{6} \rightarrow \mathbf{5}$.
3. Merge the suffixes of the 5 congruence class sets $\{0\}$, $\{1, 2, 4\}$, $\{3\}$, $\{5\}$, $\{6\}$. The necessary shift values Δ for any $x, y \in \mathcal{R}$ are:

x, y	0	1	2	3	4	5	6
0	0	1	2	1	4	4	2
1	1	0	0	1	0	3	3
2	2	0	0	6	0	6	2
3	1	1	6	0	5	6	5
4	4	0	0	5	0	4	5
5	4	3	6	6	4	0	3
6	2	3	2	5	5	3	0

2.14 C++ Implementation (DC3)

Source code excerpt from `http://www.mpi-sb.mpg.de/~sanders/programs/suffix/`:

```
// find the suffix array SA of s[0..n-1] in {1..K}^n
// require s[n]=s[n+1]=s[n+2]=0, n>=2

void suffixArray(int* s, int* SA, int n, int K) {
    int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=n0+n2;
    int* s12  = new int[n02 + 3];  s12[n02]= s12[n02+1]= s12[n02+2]=0;
    int* SA12 = new int[n02 + 3]; SA12[n02]=SA12[n02+1]=SA12[n02+2]=0;
    int* s0   = new int[n0];
    int* SA0  = new int[n0];

    // generate positions of mod 1 and mod  2 suffixes
    // the "+(n0-n1)" adds a dummy mod 1 suffix if n%3 == 1
    for (int i=0, j=0;  i < n+(n0-n1);  i++) if (i%3 != 0) s12[j++] = i;

    // lsb radix sort the mod 1 and mod 2 triples
    radixPass(s12 , SA12, s+2, n02, K);
    radixPass(SA12, s12 , s+1, n02, K);
    radixPass(s12 , SA12, s  , n02, K);

    // find lexicographic names of triples
    int name = 0, c0 = -1, c1 = -1, c2 = -1;
    for (int i = 0;  i < n02;  i++) {
        if (s[SA12[i]] != c0 || s[SA12[i]+1] != c1 || s[SA12[i]+2] != c2) {
            name++;  c0 = s[SA12[i]];  c1 = s[SA12[i]+1];  c2 = s[SA12[i]+2];
        }
        if (SA12[i] % 3 == 1) { s12[SA12[i]/3]      = name; } // left half
        else                  { s12[SA12[i]/3 + n0] = name; } // right half
    }

    // recurse if names are not yet unique
    if (name < n02) {
        suffixArray(s12, SA12, n02, name);
        // store unique names in s12 using the suffix array
        for (int i = 0;  i < n02;  i++) s12[SA12[i]] = i + 1;
    } else // generate the suffix array of s12 directly
        for (int i = 0;  i < n02;  i++) SA12[s12[i] - 1] = i;

    // stably sort the mod 0 suffixes from SA12 by their first character
    for (int i=0, j=0;  i < n02;  i++) if (SA12[i] < n0) s0[j++] = 3*SA12[i];
    radixPass(s0, SA0, s, n0, K);

    // merge sorted SA0 suffixes and sorted SA12 suffixes
    for (int p=0,  t=n0-n1,  k=0;  k < n;  k++) {
#define GetI() (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) * 3 + 2)
        int i = GetI(); // pos of current offset 12 suffix
        int j = SA0[p]; // pos of current offset 0  suffix
        if (SA12[t] < n0 ?
            leq(s[i],       s12[SA12[t] + n0], s[j],       s12[j/3]) :
            leq(s[i],s[i+1],s12[SA12[t]-n0+1], s[j],s[j+1],s12[j/3+n0]))
        { // suffix from SA12 is smaller
            SA[k] = i;  t++;
            if (t == n02) { // done --- only SA0 suffixes left
                for (k++;  p < n0;  p++, k++) SA[k] = SA0[p];
            }
        } else {
            SA[k] = j;  p++;
            if (p == n0)  { // done --- only SA12 suffixes left
                for (k++;  t < n02;  t++, k++) SA[k] = GetI();
            }
        }
    }
    delete [] s12; delete [] SA12; delete [] SA0; delete [] s0;
}
```