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:


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 $T$
- $T[i..j] := T[i]T[i+1]...T[j]$ 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 $S$. A map $\phi : S \rightarrow [0..|S|]$ is called lexicographical naming if for every $X, Y \in S$ holds: $X <_{\text{lex}} Y \iff \phi(X) < \phi(Y)$. 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 $S \subseteq \Sigma^t$ of strings having length $t$ and a lex. naming $\phi$ for $S$. Let $X_1, \ldots, X_k \in S$ and $Y_1, \ldots, Y_l \in S$ be strings from $S$. 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:

$$X_1 \cdot X_2 \cdots X_k <_{\text{lex}} Y_1 \cdot Y_2 \cdots Y_l \iff \phi(X_1)\phi(X_2)\cdots\phi(X_k) <_{\text{lex}} \phi(Y_1)\phi(Y_2)\cdots\phi(Y_l)$$

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 $i \in [n-2..n]$

\[
\begin{align*}
\tau_1 & \equiv 1 \pmod{3} \\
\tau_2 & \equiv 2 \pmod{3} \\
\tau_3 & \equiv 0 \pmod{3}
\end{align*}
\]

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

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

<table>
<thead>
<tr>
<th>$i$</th>
<th>$T[i..i+2]$</th>
<th>radix pass</th>
<th>$i$</th>
<th>$T[i..i+2]$</th>
<th>radix pass</th>
<th>$i$</th>
<th>$T[i..i+2]$</th>
<th>radix pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ACC</td>
<td>10</td>
<td>10</td>
<td>$$$</td>
<td>1</td>
<td>1</td>
<td>ACC</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CCC</td>
<td>1</td>
<td>4</td>
<td>ACC</td>
<td>5</td>
<td>2</td>
<td>CCC</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>CAC</td>
<td>2</td>
<td>7</td>
<td>CAC</td>
<td>8</td>
<td>4</td>
<td>CAC</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ACC</td>
<td>4</td>
<td>1</td>
<td>ACC</td>
<td>10</td>
<td>7</td>
<td>CAC</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>CAC</td>
<td>5</td>
<td>2</td>
<td>CCC</td>
<td>4</td>
<td>8</td>
<td>CAC</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ACC</td>
<td>7</td>
<td>5</td>
<td>ACC</td>
<td>7</td>
<td>10</td>
<td>CAC</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>$$$</td>
<td>8</td>
<td>2</td>
<td>CCC</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

$$
t_1 = \tau_1 \tau_4 \ldots \tau_{1+3n_0} \quad \text{with} \quad n_j = \left[ \frac{n-j}{3} \right]$$

$n_j$ 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 $n \equiv 1 \pmod{3} \Rightarrow n_0 - n_1 = 1$ 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** ($T = \text{GACCCACCAC}$): Construct $T' = \langle \tau_{1+3i} \mid i \in [0..n_0]\rangle \cdot \langle \tau_{2+3i} \mid i \in [0..n_2]\rangle$
\[ n = 11 \]
\[ n_0 = \left\lceil \frac{11}{3} \right\rceil = 4 \]
\[ n_2 = \left\lceil \frac{11 - 2}{3} \right\rceil = 3 \]

\[ T' = \tau_1 \tau_4 \tau_7 \tau_2 \tau_5 \tau_8 \]
\[ = 0 \ 2 \ 2 \ 1 \ 3 \ 0 \ 0 \]
\[ \equiv \text{ACC} \ \text{CAC} \ \text{CAC} \ \text{CSS} \ \text{CCC} \ \text{ACC} \ \text{ACC} \]

### 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 \( \tau_i \) are unique amongst the triples, \( A' \) can be directly be derived from \( T' \) without recursion (Exercise).

Example \( (T = \text{GACCCACC}) \):

\[ T' = 0 \ 2 \ 2 \ 1 \ 3 \ 0 \ 0 \]

\[ A'[0] = 6 \equiv 0 \equiv \text{ACC} \]
\[ A'[1] = 5 \equiv 00 \equiv \text{ACCAC} \]
\[ A'[2] = 0 \equiv 0221300 \equiv \text{ACCCACACC} \ldots \]
\[ A'[3] = 3 \equiv 1300 \equiv \text{CSS} \ldots \]
\[ A'[4] = 2 \equiv 21300 \equiv \text{CACC} \ldots \]
\[ A'[5] = 1 \equiv 221300 \equiv \text{CACC} \ldots \]
\[ A'[6] = 4 \equiv 300 \equiv \text{CCCCAC} \]

### 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 \( (T = \text{GACCCACC}) \):

\[ A'[0] = 6 \rightarrow A^{12}[0] = 8 \]
\[ A'[1] = 5 \rightarrow A^{12}[1] = 5 \]
\[ A'[2] = 0 \rightarrow A^{12}[2] = 1 \]
\[ A'[3] = 3 \rightarrow A^{12}[3] = 10 \]
\[ A'[4] = 2 \rightarrow A^{12}[4] = 7 \]
\[ A'[5] = 1 \rightarrow A^{12}[5] = 4 \]
\[ A'[6] = 4 \rightarrow A^{12}[6] = 2 \]

### 2.10 Step 2: Derive \( A^0 \) from \( A^{12} \)

Extract suffixes \( T_i \) with \( i \equiv 1 \) (mod 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

\[ T[A^0[i]] < T[A^0[j]] \text{ or } T[A^0[i]] = T[A^0[j]] \wedge T[A^0[i] + 1..n - 1] <_{\text{lex}} T[A^0[j] + 1..n - 1] \text{ holds.} \]
**Example ($T = \text{GACCCACCACC}$):**

$A^{12} = 8 \ 5 \ 1 \ 10 \ 7 \ 4 \ 2$

$A^0 = \ 0 \ 9 \ 6 \ 3$

$A^0[0] = 0 \equiv \text{GACCCACCACC}$ \xrightarrow{\text{rank}} \ A^0[0] = 9 \equiv \text{CC}$

$A^0[1] = 9 \equiv \text{CC}$ \xrightarrow{\text{pass}} \ A^0[1] = 6 \equiv \text{CCACC}$

$A^0[2] = 6 \equiv \text{CCACC}$ \xrightarrow{\text{pass}} \ A^0[2] = 3 \equiv \text{CCACCACC}$

$A^0[3] = 3 \equiv \text{CCACCACC}$ \xrightarrow{\text{pass}} \ A^0[3] = 0 \equiv \text{GACCCACCACC}$

### 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 $n \equiv 1 \ (\mod \ 3)$, 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:** $i \equiv 0 \ (\mod \ 3)$ and $j \equiv 1 \ (\mod \ 3)$

$$T[i..n-1] <_{\text{lex}} T[j..n-1] \Leftrightarrow \begin{cases} \left( T[i] < T[j] \right) \lor \\
\left( T[i] = T[j] \land R^{12}[i+1] < R^{12}[j+1] \right) \end{cases}$$

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

**Case 2:** $i \equiv 0 \ (\mod \ 3)$ and $j \equiv 2 \ (\mod \ 3)$

$$T[i..n-1] <_{\text{lex}} T[j..n-1] \Leftrightarrow \begin{cases} \left( T[i..i+1] <_{\text{lex}} T[j..j+1] \right) \lor \\
\left( T[i..i+1] =_{\text{lex}} T[j..j+1] \land R^{12}[i+2] < R^{12}[j+2] \right) \end{cases}$$

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

**Example ($T = \text{GACCCACCACC}$):**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$\text{G} \ \text{A} \ \text{C} \ \text{C} \ \text{A} \ \text{C} \ \text{C} \ \text{C} \ \text{A} \ \text{C} \ \text{C} \ \text{A} \ \text{C}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$R^{12}$</td>
<td>3</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$A^{12} = 8 \ 5 \ 1 \ 10 \ 7 \ 4 \ 2$

$A^0 = \ 0 \ 9 \ 6 \ 3 \ 0$

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

Compare $T_8$ with $T_9$:

$T[8..9] = \text{AC} <_{\text{lex}} \text{CC} = T[9..10] \Rightarrow A[0] = 8$

$A = 8$

$A^{12} = 8 \ 5 \ 1 \ 10 \ 7 \ 4 \ 2$

$A^0 = \ 0 \ 9 \ 6 \ 3 \ 0$

Compare $T_5$ with $T_9$:

$T[5..6] = \text{AC} <_{\text{lex}} \text{CC} = T[9..10] \Rightarrow A[1] = 5$

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

Compare $T_1$ with $T_9$:

$A = \begin{array}{cccc} 8 & 5 & 1 & \end{array}$

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

Compare $T_{10}$ with $T_9$:
$T[10] = C = C = T[9] \ \wedge \\

$A = \begin{array}{ccccccc} 8 & 5 & 1 & 10 & \end{array}$

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

Compare $T_7$ with $T_9$:
$T[7] = C = C = T[9] \ \wedge \\

$A = \begin{array}{ccccccc} 8 & 5 & 1 & 10 & 7 & \end{array}$

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

Compare $T_4$ with $T_9$:

$A = \begin{array}{ccccccc} 8 & 5 & 1 & 10 & 7 & 4 & \end{array}$

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

Compare $T_2$ with $T_9$:
$T[2..3] = CC = \text{uns} \ CC = T[9..10] \ \wedge \\

$A = \begin{array}{ccccccc} 8 & 5 & 1 & 10 & 7 & 4 & 9 & \end{array}$

$A^{12} = \begin{array}{ccccccc} 8 & 5 & 1 & 10 & 7 & 4 & 2 \\ A^0 = \end{array}$
Compare $T_2$ with $T_6$:

\[
\begin{align*}
T[2..3] &= \text{CC} \stackrel{\text{eq}}{=} \text{CC} = T[6..7] \land \\
\end{align*}
\]

\[
\begin{align*}
A &= 8 \ 5 \ 1 \ 10 \ 7 \ 4 \ 9 \ 6 \\
A^{12} &= 8 \ 5 \ 1 \ 10 \ 7 \ 4 \ 2 \\
A^0 &= 9 \ 6 \ 3 \ 0
\end{align*}
\]

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

\[
A = 8 \ 5 \ 1 \ 10 \ 7 \ 4 \ 9 \ 6 \ 3 \ 2 \ 0
\]

Done. The resulting suffix array is:

\[
\begin{align*}
A[0] &= 8 \equiv \text{ACC} \\
A[1] &= 5 \equiv \text{ACCACC} \\
A[2] &= 1 \equiv \text{ACCCACCACC} \\
A[3] &= 10 \equiv \text{C} \\
A[4] &= 7 \equiv \text{CACC} \\
A[5] &= 4 \equiv \text{CACCACC} \\
A[6] &= 9 \equiv \text{CC} \\
A[7] &= 6 \equiv \text{CCACC} \\
A[8] &= 3 \equiv \text{CCACCACC} \\
A[9] &= 2 \equiv \text{CCACCACC} \\
A[10] &= 0 \equiv \text{GACCACCACC}
\end{align*}
\]

### 2.12 Linear running time

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

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

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

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

**Proof:** Exercise.
2.13 Difference Covers

The key idea of the skew algorithm is to

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

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

The recursion depth of the skew algorithm heavily depends on $\lambda = \frac{|\mathcal{R}|}{|I|}$ the factor the text length decreases with. Is it possible to find $I$ and $\mathcal{R}$ yielding a smaller $\lambda$ 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 $I$ to be difference cover if for any $z \in \mathcal{R}$ there exist $a, b \in I$ such that $a - b = z$.

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

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

$$(x + \Delta) = x + (a - x) = a \Rightarrow (x + \Delta) \in I$$

and

$$(y + \Delta) = y + (a - x) = a - (x - y) = a - z = b \Rightarrow (y + \Delta) \in I$$

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

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

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

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

A next greater perfect difference cover is $I = \{1 + 7\mathbb{Z}, 2 + 7\mathbb{Z}, 4 + 7\mathbb{Z}\}$ 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: $4 \rightarrow 3$ and $1 \rightarrow 0 \rightarrow 6 \rightarrow 5$.
3. Merge the suffixes of the $5$ congruence class sets $\{0\}, \{1, 2, 4\}, \{3\}, \{5\}, \{6\}$. The necessary shift values $\Delta$ for any $x, y \in \mathcal{R}$ are:

$$
\begin{array}{cccccccc}
x, y & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
0 & 0 & 1 & 2 & 1 & 4 & 4 & 2 \\
1 & 1 & 0 & 0 & 1 & 0 & 3 & 3 \\
2 & 2 & 0 & 0 & 6 & 0 & 6 & 2 \\
3 & 3 & 1 & 1 & 0 & 5 & 6 & 5 \\
4 & 4 & 0 & 0 & 5 & 0 & 4 & 5 \\
5 & 5 & 4 & 3 & 6 & 6 & 4 & 0 \\
6 & 6 & 2 & 3 & 2 & 5 & 3 & 0
\end{array}
$$

```c++
// 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++
```