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:

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

**Example 4.20:** $D_7 = \{1, 2, 4\}$

- $1 - 1 = 0 \quad 1 - 4 = -3 \equiv 4 \pmod q$
- $2 - 1 = 1 \quad 2 - 4 = -2 \equiv 5 \pmod q$
- $4 - 2 = 2 \quad 1 - 2 = -1 \equiv 6 \pmod q$
- $4 - 1 = 3$

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 $O(\sqrt{q})$.
- 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

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

**Example 4.21:** If $T = \text{banana}\$ and $D_3 = \{1, 2\}$, then $C = \{1, 2, 4, 5\}$ and $T_C = \{\text{anana}\$, $\text{nana}\$, $\text{na}\$, $\text{a}\$\}$. 

Once we have sorted the difference cover sample $T_C$, we can compare any two suffixes in $O(q)$ 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 $\ell$ 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 4.22:** $D_3 = \{1, 2\}$ and $C = \{1, 2, 4, 5, \ldots\}$

$$T_0 = T[0]T_1$$
$$T_1 = T[1]T_2$$
$$T_3 = T[3]T_4$$
Algorithm 4.23: DC3

Step 0: Choose $C$.

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

Example 4.24:

<table>
<thead>
<tr>
<th>i</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>$T[i]$</td>
<td>y</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>d</td>
<td>o</td>
<td>$ $</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\bar{C} = C_0 = \{0, 3, 6, 9, 12\}$, $C_1 = \{1, 4, 7, 10\}$, $C_2 = \{2, 5, 8, 11\}$ 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 a lexicographic name $N^3_i \in [1..|R|]$. The names can be computed by sorting the factors with LSD radix sort in $O(n)$ 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 lexicographic 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 $i \in \overline{C}$, let $N_i = \perp$. Also let $N_{n+1} = N_{n+2} = 0$.

**Example 4.25:**

<table>
<thead>
<tr>
<th>$R$</th>
<th>abb</th>
<th>ada</th>
<th>bba</th>
<th>do$</th>
<th>bba</th>
<th>dab</th>
<th>bad</th>
<th>o$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R'$</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>$SA^{−1}_{R'}$</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

- $i$ | 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 | $|

$N_i$ | $\perp$ | 1 | 4 | $\perp$ | 2 | 6 | $\perp$ | 5 | 3 | $\perp$ | 7 | 8 | $\perp$ | 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 4.26:

<table>
<thead>
<tr>
<th>$i$</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>$T[i]$</td>
<td>y</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>d</td>
<td>o</td>
<td>$</td>
</tr>
<tr>
<td>$N_i$</td>
<td>$\bot$</td>
<td>1</td>
<td>4</td>
<td>$\bot$</td>
<td>2</td>
<td>6</td>
<td>$\bot$</td>
<td>5</td>
<td>3</td>
<td>$\bot$</td>
<td>7</td>
<td>8</td>
<td>$\bot$</td>
</tr>
</tbody>
</table>

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

195
Step 2(b): Merge $T_C$ and $T_{\bar{C}}$.

- Use comparison based merging algorithm needing $O(n)$ comparisons.

- To compare $T_i \in T_C$ and $T_j \in T_{\bar{C}}$, we have two cases:
  
  $i \in C_1: \quad T_i \leq T_j \iff (T[i], N_{i+1}) \leq (T[j], N_{j+1})$
  
  $i \in C_2: \quad T_i \leq T_j \iff (T[i], T[i+1], N_{i+2}) \leq (T[j], T[j+1], N_{j+2})$

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

Example 4.27:

\[
\begin{array}{cccccccccccc}
  i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
  T[i] & y & a & b & b & a & d & a & b & b & a & d & o & $ \\
  N_i & \bot & 1 & 4 & \bot & 2 & 6 & \bot & 5 & 3 & \bot & 7 & 8 & \bot \\
\end{array}
\]

$T_1 < T_6$ because $(a, 4) < (a, 5)$.

$T_3 < T_8$ because $(b, a, 6) < (b, a, 7)$. 
Theorem 4.28: Algorithm DC3 constructs the suffix array of a string $T[0..n)$ in $O(n)$ 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 $O(n \log n)$ and the space needed in addition to the text and the suffix array is $O(n/\sqrt{\log n})$. 
**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 4.29:**

<table>
<thead>
<tr>
<th>$i$</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>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T[i]$</td>
<td>m</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>i</td>
<td>$</td>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>

Type of $T_i$ $-$ $-$ $*$ $-$ $-$ $*$ $-$ $-$ $*$ $+$ $-$ $-$ $-$ $-$

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

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

Then

- $C^-_a = \{ i \in C_a \mid T_i < a^{n+1} \}$
- $C^+_a = \{ i \in C_a \mid T_i > a^{n+1} \}$

and thus, if $i \in C^-_a$ and $j \in C^+_a$, then $T_i < T_j$. Hence the suffix array is $C_0C_1C_2 \ldots C_{\sigma-1} = C_0C^-_1C^+_1C^-_2C^+_2 \ldots C^-_{\sigma-1}C^+_{\sigma-1}$. 

198
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 4.30:** 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. $i \in C_0$ ($i = n$)
   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 $T[i - 1] \leq T[i]$. 

199
To induce $\cdash$-type 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 4.30(a), Step 2 can be done by checking the relevant conditions for all $i \in C_0 C_1^-C_1^*C_2^-C_2^*\ldots$.

**Algorithm 4.31:** 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 $i - 1$. 

200
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 4.32**: InducePlusSuffixes

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) for $a \leftarrow \sigma - 1$ downto 1 do
(3) for $i \in C^+_a$ in reverse order do // include elements added during loop
(4) if $i > 0$ and $T[i - 1] \leq a$ then pushfront($i - 1, C^+_{T[i-1]}$)
(5) for $i \in C^-_a$ in reverse order do
(6) 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 \( \sigma \) is a special symbol larger than any other symbol.

**Lemma 4.33:** 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 4.30, \( T[\ell] < T[\ell - 1] \).
- \( T[k - 1..k] = T[\ell - 1..\ell] \) and thus \( T[k] < T[k - 1] \). 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 \( T_k < a^{n+1} < T_\ell \).
- Since \( T[i..k] = T[j..\ell] \) and \( T_k < T_\ell \), we have \( T_i < T_j \).

\( \square \)
Algorithm 4.34: SAIS

Step 0: Choose $C$.

- Compute the types of suffixes. This can be done in $O(n)$ time based on Lemma 4.30.
- Set $C = \bigcup_{a \in [1..\sigma)} C^*_a \cup \{n\}$. Note that $|C| \leq n/2$, since for all $i \in C$, $i - 1 \in C^- \subseteq C$.

Example 4.35:

\begin{center}
\begin{tabular}{cccccccccccccc}
  $i$ & 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 $T_i$ & $-$ & $-$ & * & $-$ & $-$ & * & $-$ & $-$ & * & $+$ & $-$ & $-$ & $-$ & $-$
\end{tabular}
\end{center}

$C^*_i = \{2, 5, 8\}$, $C^*_m = C^*_p = C^*_s = \emptyset$, $C = \{2, 5, 8, 14\}$.
Step 1: Sort $T_C$.

- Sort the strings $S'_i, i \in C^*$. Since the total length of the strings $S'_i$ is $O(n)$, the sorting can be done in $O(n)$ time using LSD radix sort.
- Assign lexicographic 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 $R = N_{i_1}N_{i_2} \ldots N_{i_k}0$, where $i_1 < i_2 < \cdots < i_k$ 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 4.36:

<table>
<thead>
<tr>
<th>$i$</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>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T[i]$</td>
<td>m</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>$N_i$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$R = [issiz][issiz][iippii$z$]$ = 2210, $SA_R = (3, 2, 1, 0)$, $C^*_1 = (8, 5, 2)$
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 $SA = nC_1^-C_1^+C_2^-C_2^+\ldots C_{\sigma-1}^-C_{\sigma-1}^+.$

Example 4.37:

<table>
<thead>
<tr>
<th>$i$</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>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T[i]$</td>
<td>m</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>i</td>
<td>$</td>
</tr>
<tr>
<td>type of $T_i$</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>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

$C_\$ = (14) $\Rightarrow$ $C_i^- = (13, 12)$

$C_i^-C_i^* = (13, 12, 8, 5, 2)$ $\Rightarrow$ $C_m^- = (1, 0)$, $C_p^- = (11, 10)$, $C_s^- = (7, 4, 6, 3)$

$\Rightarrow$ $C_i^+ = (8, 9, 5, 2)$

$\Rightarrow$ $SA = C_\$C_i^-C_i^+C_m^-C_p^-C_s^- = (14, 13, 12, 8, 9, 5, 2, 1, 0, 11, 10, 7, 4, 6, 3)$
Theorem 4.38: Algorithm SAIS constructs the suffix array of a string $T[0..n)$ in $O(n)$ 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 $C^+_1C^+_2\ldots C^+_{\sigma-1}$.

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 $O(n \log n)$ time and then continues as SAIS.