In backward search, we need to compute the range \([b_i, e_i)\) from the range \([b_{i+1}, e_{i+1})\). This is done separately for each end of the range.

Given \(b_{i+1}\), we can compute \(b_i\) as follows.

- Recall that \(b_i\) is the first row in \(M\) beginning with \(P_i = P[1..m]\), i.e., the number of rows that are lexicographically smaller than \(P_i\).
- \(C[P[i]]\) is the number of rows beginning with a symbol smaller than \(P[i]\).
- To \(C[P[i]]\) we need to add the number of rows that begin with \(P[i]\) and are lexicographically smaller than \(P_i\).
- \(\text{rank}_L(P[i], b_{i+1})\) is the number of rows that are lexicographically smaller than \(P_{i+1}\) and contain \(P[i]\) at the last column. Rotating these rows one step to the right, we obtain the rotations of \(T\) that begin with \(P[i]\) and are lexicographically smaller than \(P[T[i]] = P_i\).
- Thus \(b_i = C[P[i]] + \text{rank}_L(P[i], b_{i+1})\).

Computing \(e_i\) from \(e_{i+1}\) is similar: \(e_i = C[P[i]] + \text{rank}_L(P[i], e_{i+1})\).

**Algorithm 4.15: Backward Search**

**Input:** array \(C\), function \(\text{rank}_L\), pattern \(P\)

**Output:** suffix array range \([b..e)\) containing starting positions of \(P\)

1. \(b ← 0\), \(e ← n + 1\)
2. for \(i ← m−1\) downto 0 do
   3. \(c ← P[i]\)
   4. \(b ← C[c] + \text{rank}_L(c, b)\)
   5. \(e ← C[c] + \text{rank}_L(c, e)\)
6. return \([b..e)\)

- The array \(C\) requires an integer alphabet that is not too large.
- The trivial implementation of the function \(\text{rank}_L\), as an array requires \(\Theta(n^2)\) space, which is often too much. There are much more space efficient (but slower) implementations. There are even implementations with a size that is close to the size of the compressed text. Such an implementation is the key component in many compressed text indexes. These are covered in the course Data Compression Techniques.

**LCP Array Construction**

The LCP array is easy to compute in linear time from the suffix array with the help of a couple of auxiliary arrays:

- For each \(i \in [1..n]\), let \(Φ[SA[i]] = 0\). \(j\). Then the suffix \(T_{Φ[i]}\) is the immediate lexicographical predecessor of the suffix \(T_j\).
- For each \(i \in [1..n]\), let \(\text{PLCP}[SA[i]] = LCP[P[i]]\). Then \(\text{PLCP}[j] = LCP[SA[j]]\). \(i..m\), \(PLCP[i]\) is the lcp between \(T_j\) and its lexicographical predecessor.

**Example 4.16:**

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
i \quad | \quad SA[i] & LCP[i] & T_{SA[i]} & j \quad | \quad SA[j] & Φ[j] & PLCP[j] & T_j \\
\hline
0 & 6 & 0 & 0 & 6 & 0 & banana$ \\
1 & 5 & 0 & 0 & 3 & 3 & 3 \quad Φ[1] \\
2 & 3 & 1 & 3 & 2 & 2 & banana$ \\
3 & 3 & 1 & 3 & 2 & 2 \quad anana$ \\
4 & 0 & 0 & 0 & 4 & 4 & 2 \quad banana$ \\
5 & 4 & 0 & 0 & 5 & 5 & 2 \quad anana$ \\
6 & 2 & 2 & 2 & 6 & 6 & 0 \quad a$ \\
\hline
\end{array}
\]

The algorithm computes first \(Φ\) then \(\text{PLCP}\) and finally \(LCP\). The computation of \(\text{PLCP}\) takes advantage of the above lemma.

**Algorithm 4.18:**

**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. for \(i \in [1..n]\) do \(Φ[SA[i]] ← SA[i]−1\)
2. \(ℓ ← 0\)
3. for \(y ← 0\) to \(n−1\) do
   4. \(while T[ℓ] + 1 = T[Φ[ℓ] + 1] do \(ℓ ← ℓ + 1\)\)
   5. \(\text{PLCP}[ℓ] ← ℓ\)
   6. \(if ℓ > 0 \ then \(ℓ ← ℓ−1\)\)
7. for \(i \in [1..n]\) do \(LCP[i] ← \text{PLCP}[SA[i]]\)
8. return \(LCP\)

The time complexity is \(O(n)\) in the general alphabet model:

- Everything except the while loop on line (4) takes clearly linear time.
- Each round in the loop increments \(ℓ\). Since \(ℓ\) is decremented at most \(n\) times on line (6) and cannot grow larger than \(n\), the loop is executed \(O(n)\) times in total.

**Prefix Doubling**

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

Let \(T^i\) denote the text factor \(T[i..\min(i + ℓ, n + 1)]\) and call it an \(ℓ\)-factor. In other words:

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

The idea is to sort the sets \(T^i_{[0..n]}\) for ever increasing values of \(ℓ\).

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

**Suffix Array Construction**

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

- Using standard sorting or string sorting the time complexity is \(Ω(L\text{LCP}(T[0..n]))\).
- 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 \(O(n)\) in the constant alphabet model.

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

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

\[N^i < N^j\ if \ and \ only \ if \ T^i < T^j\ ;\]

Then, for \(ℓ > n\, N^i = SA^{-1}[i]\).

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 \(N^i = \{j \in [0..n] \mid T^j < T^i\}\).

**Example 4.19:**

**Prefix doubling for \(T = \text{banana}$\)**

\[
\begin{array}{|c|c|c|c|c|}
\hline
N^1 & N^2 & N^3 & N^4 \\
\hline
4 & 6 & 3 & 7 \quad \text{banana} \\
3 & 6 & 3 & 7 \quad \text{banana} \\
1 & 7 & 2 & 2 \quad \text{nana} \\
6 & 1 & 2 & 7 \quad \text{nana} \\
7 & 6 & 5 & 1 \quad \text{nana} \\
1 & 5 & 3 & 5 \quad \text{nana} \\
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
U & U & U & U \\
1 & 3 & 1 & 4 \\
\end{array}
\]
Now, given $N_\ell$, for the purpose of sorting, we can use
• $N_\ell i$ to represent $T_\ell i$
• the pair $(N_\ell i, N_\ell i+\ell)$ to represent $T_\ell i T_\ell i+\ell$.
Thus we can sort $T_{[0..n]}$ by sorting pairs of integers, which can be done in $O(n \log n)$ time using LSD radix sort.

Theorem 4.20: The suffix array of a string $T[0..n]$ can be constructed in $O(n \log n)$ 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 variant is the Larson–Sadakane algorithm, which uses ternary quicksort instead of LSD radix sort for sorting the pairs, but still achieves $O(n \log n)$ total time.

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 $T[n] = \$$ (as 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 $T_C$.

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

• In the general alphabet model, we can use ternary quicksort for time complexity $O(n \log \sigma_T)$ where $\sigma_T$ is the number of distinct symbols in $T$.
• In the integer alphabet model with $\sigma = O(n^c)$ for any constant $c$, we can use LSD radix sort with radix $\alpha$ for time complexity $O(n \alpha)$.

After this, we can replace each character $T[i]$ with $N',i$ 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 construct the suffix array using $T'$ as the text instead of $T$. As we will see next, the suffix array of $T'$ can be constructed in linear time. Then sorting the characters of $T'$ to obtain $T''$ is the asymptotically most expensive operation in the suffix array construction of $T$ for any alphabet.

Assume that
• $|C| \le \alpha n$ 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 $t(n) = O(n) + t(\alpha n)$, whose solution is $t(n) = O(n)$.

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 covering sampling
• SAIS uses induced sorting

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

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

Example 4.22: If $T = \text{banana} \$$ and $D_3 = \{1, 2, 4, 5, \ldots \}$

Then $C = \{\text{banana} \}$ and $T_{[0..n]} = \{\text{banana} \}$. We can construct the suffix array of $T = \text{banana} \$$.

A difference cover sample is a set of suffixes, which is easily within the time bound $O(n \log \sigma_T)$ where $\sigma_T$ is the number of distinct symbols in $T$.

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 $T[n] = \$$ (as 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 $T_C$.

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 4.21: $D_3 = \{1, 2, 4\}$

<table>
<thead>
<tr>
<th>$i - j$</th>
<th>$i - j \mod 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 1</td>
<td>0</td>
</tr>
<tr>
<td>1 - 2</td>
<td>1</td>
</tr>
<tr>
<td>1 - 4</td>
<td>-2 = 1</td>
</tr>
<tr>
<td>4 - 2</td>
<td>2</td>
</tr>
<tr>
<td>4 - 4</td>
<td>0</td>
</tr>
</tbody>
</table>

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 $D_3 = \{1, 2\}$.

Algorithm 4.24: 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 $C = C_0 \cup C_1$ and $C = C_0$.

Example 4.25:

<table>
<thead>
<tr>
<th>$T[i]$</th>
<th>$y$</th>
<th>$a$</th>
<th>$b$</th>
<th>$b$</th>
<th>$a$</th>
<th>$b$</th>
<th>$d$</th>
<th>$a$</th>
<th>$b$</th>
<th>$b$</th>
<th>$d$</th>
<th>$o$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_0$</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C_1$</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C_2$</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<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>
</tr>
</thead>
<tbody>
<tr>
<td>$T[i]$</td>
<td>$y$</td>
<td>$a$</td>
<td>$b$</td>
<td>$b$</td>
<td>$a$</td>
<td>$b$</td>
<td>$d$</td>
<td>$a$</td>
<td>$b$</td>
<td>$b$</td>
<td>$d$</td>
<td>$o$</td>
</tr>
<tr>
<td>$N_1$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

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 character $T_\ell i$ in $R$ with an order preserving name $N_\ell i \in [1..|R|]$. The names can be computed by sorting the characters with LSD radix sort in $O(n \log \sigma_T)$ time. Let $R' = R$ be the result appended with 0.

Construct the inverse suffix array $SA_{R'}$. This is done recursively using DC3 unless all symbols in $R'$ are unique, in which case $SA_{R'} = R'$.

From $SA_{R'}$, we get order preserving names for suffixes in $T_C$.

For $i \in C$, let $N_1 i = SA_{R'}[i]$, where $j$ is the position of $T_\ell i$ in $R$. For $i \in C$, let $N_{1+} i = N_{1+} 0$. Also let $N_{1+} 0 = 0$.

Example 4.26:

<table>
<thead>
<tr>
<th>$R$</th>
<th>$R$</th>
<th>$R$</th>
<th>$R$</th>
<th>$R$</th>
<th>$R$</th>
<th>$R$</th>
<th>$R$</th>
<th>$R$</th>
<th>$R$</th>
<th>$R$</th>
<th>$R$</th>
<th>$R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>$0$</td>
<td>$1$</td>
<td>$2$</td>
<td>$3$</td>
<td>$4$</td>
<td>$5$</td>
<td>$6$</td>
<td>$7$</td>
<td>$8$</td>
<td>$9$</td>
<td>$10$</td>
<td>$11$</td>
</tr>
<tr>
<td>$T[i]$</td>
<td>$y$</td>
<td>$a$</td>
<td>$b$</td>
<td>$b$</td>
<td>$a$</td>
<td>$b$</td>
<td>$d$</td>
<td>$a$</td>
<td>$b$</td>
<td>$b$</td>
<td>$d$</td>
<td>$o$</td>
</tr>
<tr>
<td>$N_0$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
**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 $i \in \bar{C}$.

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

**Example 4.27:**

<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>$N_i$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td></td>
</tr>
</tbody>
</table>

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

**Theorem 4.29:** 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 $O(n / \sqrt{\log n})$.

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

$C_x = \{i \in [0..n] \mid T[i] = a\}$

$C^+ = C_+ \cap C^-$

The two types of suffixes starting with the same character are lexigraphically separated.

**Lemma 4.31:** For all $a \in \Sigma$, we have $C_+ \in C^-$. Hence the suffix array is $aC_1C_2\ldots C_{n-1} = aC_1^+C_2^+\ldots C_{n-1}^+C_{n-2}^-$.

To induce $C^-$ suffixes:

1. Set $C_0^-$ 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 4.32(a), Step 2 can be done by checking the relevant conditions for all $i \in aC_1^+C_2^+\ldots C_{n-1}^+C_{n-2}^-\ldots$.

**Algorithm 4.33:** InduceMinusSuffixes

Input: Lexicographically sorted lists $C_1^+, a \in \Sigma$.

Output: $C_0^-$

1. for $a \in \Sigma$ do $C_a^- = \emptyset$
2. pushback($n-1$, $C_{T[n-1]}^-$)
3. for $i = n-1$ downto 1 do
4. if $i > 0$ and $T[i] \geq a$ then pushback($i-1$, $C_{T[i-1]}^-$)
5. 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 $i-1$.

**Step 2(b):** Merge $T_i$ and $T_j$.

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

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

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

**Example 4.28:**

<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>$N_i$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td></td>
</tr>
</tbody>
</table>

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

**Induced Sorting**

Define two types of suffixes, $-$ and $+$, as follows:

$C^- = \{i \in [0..n] \mid T[i] > T_{i+1}\}$

$C^+ = \{i \in [0..n] \mid T[i] < T_{i+1}\}$

Furthermore, for each run of consecutive suffixes of the same type, define the leftmost suffix as $a + suffix$:

$C_a^- = \{i \in C^- \mid i-1 \in C^+\}$

$C_a^+ = \{i \in C^+ \mid i-1 \in C^-\}$

**Example 4.30:**

<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>$a$</td>
<td>$d$</td>
<td>$a$</td>
<td>$b$</td>
<td>$a$</td>
<td>$b$</td>
<td>$d$</td>
<td>$a$</td>
<td>$b$</td>
<td>$d$</td>
</tr>
</tbody>
</table>

$T_6 < T_9$ because $(a, 4) < (a, 5)$. $T_3 < T_8$ because $(b, a, 6) < (b, a, 7)$.

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 $C_a^+ \subseteq C_a^-$, $a \in [1..\sigma)$, 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 4.32:** For all $a \in [1..\sigma)$:

1. $i = n$.
2. $i \in C_a^+$.
3. $i \in C^-$ and $T[i-1] > T[i]$.

**Algorithm 4.34:** InducePlusSuffixes

Input: Lexicographically sorted lists $C_a^+$, $a \in \Sigma$.

Output: $C_a^+$

1. for $a \in [1..\sigma)$ do $C_a^+ = \emptyset$
2. for $a \in [1..\sigma)$ do $C_a^+ = \emptyset$
3. for $i \in C_a^+$ in reverse order do $i \leftarrow i - 1$ downto 1 do
4. if $i > 0$ and $T[i-1] \leq a$ then pushfront($i-1$, $C_{T[i-1]}^-$)
5. for $i \in C_a^+$ do pushfront($i-1$, $C_{T[i-1]}^-$)
We still need to explain how to sort the $-\ast$-type suffixes. For this we need the following definition and result:

$$F[i] = \min\{k \in [i + 1..n] \mid k \in C^\ast \text{ or } k = n\}$$

S$\{0\}$ = $T[i].F[i]$

**Lemma 4.35:** For any $i, j \in [0..n]$, $T_i < T_j$ iff $S_i < S_j$ or $S_i = S_j$ and $T_i < T_j \mid F(i) < F(j)$.

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

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

- $k \in C^\ast$ and thus $k - 1 \in C^\ast$. Since $T_i < a < T_{k-1}$ by Lemma 4.31, we must have $T[k-1] \succ T[i]$.
- $T[i..k) = T[j..\ell)$ and thus $T[k-1] \succ T[i]$. If we had $\ell \in C^\ast$, we would have $T[k-1] \prec C^\ast$. Since this is not the case, we must have $\ell \in C^\ast$.
- Since $k \in C^\ast$ and $\ell \in C^\ast$, we must have $T_k < a < T_{\ell}$.
- Since $T[i..k) = T[j..\ell)$ and $T_k < T_{\ell}$, we have $T_i < T_j$.

\[ \square \]

**Algorithm 4.36: SAIS**

**Step 0:** Choose $C$.

- Compute the types of suffixes. This can be done in $O(n)$ time based on Lemma 4.32.
- Set $C = \cup_{a \in [1..\sigma]} C^+_a$. Note that $|C| \leq n/2$, since for all $i \in C$, $i - 1 \in C^+ \subseteq C$.

**Example 4.37:**

- $T[i] = [a b b a d a b b a d o \$]
  - Type of $T_i = + - + - + + - + + +$

**Step 2:** Sort $T[i..n]$.

- Run InducePlusSuffixes to construct the sorted lists $C^+_a$, $a \in [1..\sigma]$.
- Run InduceMinusSuffixes to construct the sorted lists $C^-_a$, $a \in [1..\sigma]$.

- The suffix array is $S_A = nC^+_1 C^+_2 C^+_3 \ldots C^+_n C^-_1 C^-_2 C^-_3 \ldots$. In fact, $n \sigma = (12, 1, 3, 4, 9, 7, 2, 8, 6, 5, 11, 10)$.

**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 models.

Often augmented with additional data structures:

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

More and more often suffix trees and arrays are replaced by compressed text indexes, often based on the BWT.

**Selected Literature**

**Survey**


**Suffix tree construction**


**Suffix array**


**Enhanced suffix array**


• Burrows–Wheeler transform


• Backward search


• LCP array construction


• Suffix array construction: survey


• Suffix array construction: prefix doubling


• Suffix array construction: difference cover sampling


• Suffix array construction: induced sorting
