Let us return to the first phase of the prefix doubling algorithm: assigning names \( N_i \) 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:

- On an ordered alphabet, we can use ternary quicksort for time complexity \( O(n \log \sigma_T) \) where \( \sigma_T \) is the number of distinct symbols in \( T \).
- On an integer alphabet of size \( n^\alpha \) for any constant \( \alpha \), we can use LSD radix sort with radix \( n^\alpha \) for time complexity \( O(n) \).

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 \( T'[0] = 0 \) is the unique, smallest symbol, i.e., \( \perp \).
- 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.

### Difference Cover Sampling

A difference cover \( D_q \) modulo \( q \) is a subset of \([0 \ldots q)\) such that all values in \([0..q)\) can be represented as \( i \equiv j \mod q \) for any \( i, j \in [0..q) \).

Example 4.21: If \( T = \text{banana}\$ \) and \( D_3 = \{1,2\} \), then \( C = \{i \in [0..n] \mid (i \mod 3) \in D_3\} \).

\[
\begin{align*}
T_0 &= T[0]T_1 \\
T_1 &= T[1]T_2 \\
T_3 &= T[3]T_4
\end{align*}
\]

The set \( C \) must be chosen so that:

1. Sorting \( T_C \) can be reduced to suffix array construction on a text of length \( |C| \).
2. Given sorted \( T_C \), the suffix array of \( T \) is easy to construct.

We look at two different ways of choosing \( C \) leading to two different algorithms:

- DC3 uses difference cover sampling
- SAIS uses induced sorting

### Recursive Suffix Array Construction

Let us now describe linear time algorithms for suffix array construction. We assume that the alphabet of the text \( T[0..n] = [1..n] \) and that \( T[0] = 0 \) (as in the examples).

The outline of the algorithms is:

1. Choose a subset \( C \subseteq [0..n] \).
2. Sort the set \( T_C \). This is done by a reduction to the suffix array construction of a string of length \( |C| \), which is done recursively.

The set \( C \) can be chosen so that:

- \( |C| \leq n \alpha \) for a constant \( \alpha < 1 \).
- Excluding the recursive call, all steps can be done in linear time.

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

### 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 \( D_q \).

In other words:

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

Example 4.20: \( D_4 = \{1,2,4\} \)

\[
\begin{align*}
1 &- 1 = 0 \\
1 &- 4 = -3 = 4 \mod q \\
2 &- 1 = 1 \\
2 &- 4 = -2 = 5 \mod q \\
4 &- 2 = 2 \\
4 &- 1 = 3 \\
4 &- 4 = 0
\end{align*}
\]

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 4.23: DC3

**Step 0:** Choose \( C \).

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

**Example 4.24:** \( i = 0 \ 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12 \)

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

**Step 1:** Sort \( T_C \).

- For \( k \in [1,2] \), construct the strings \( R_k = \{T[0], T[3], T[6], \ldots, T[3k-3], T[3k] \} \) whose characters are 3-factors of the text, and let \( R = R_0 \cup R_2 \).
- Replace each factor \( T'_3 \) in \( R \) with an order preserving name \( N^3 \in [1..|R|] \). The names can be computed by sorting the factors with LSD radix sort in \( O(n) \) time. Let \( R' \) be the result with appended 0.
- Construct the inverse suffix array \( S_{A3}^R \) of \( R' \). This is done recursively using DC3 unless all symbols in \( R' \) are unique, in which case \( S_{A3}^R = R' \).
- From \( S_{A3}^R \), we get order preserving names for suffixes in \( T_C \).

**Example 4.25:**

| \( R \) | \begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\end{array} \\
| \begin{array}{cccccccccccc}
y & a & b & b & a & d & b & b & a & d \ \$ \\
\end{array} \\
S_{A3}^R & \begin{array}{cccccccccccc}
1 & 2 & 5 & 7 & 4 & 6 & 3 & 8 & 0 \\
\end{array} \\
N_i & \begin{array}{cccccccccccc}
1 & 1 & 4 & 2 & 6 & 5 & 3 & 7 & 8 & 0 \\
\end{array}
|
Step 2(b): Merge $T_C$ and $T_{\overline{C}}$.

- Use comparison based merging algorithm needing $O(n)$ comparisons.
- To compare $T_i \in T_C$ and $T_{\overline{i}} \in T_{\overline{C}}$, we have two cases:
  \begin{align*}
  i &\in C_1 \land T_i \preceq T[i], N_{i-1} \preceq \langle T[i], N_{i+1} \rangle \\
  i &\in C_2 \land T_i \preceq T[i], T[i+1], N_{i+2} \preceq \langle T[i], T[i+1], N_{i+2} \rangle
  \end{align*}
Note that none of the $N$-values is $\perp$.

Example 4.27:
\[
\begin{array}{c}
T[i] = a \ b \ a \ d \ a \ b \ b \ a \ d \ o \\
N_i = 1 \ 4 \ 1 \ 2 \ 6 \ 5 \ 3 \ 1 \ 7 \ 8 \ 1
\end{array}
\]
$T_1 < T_6$ because $(a, 4) < (a, 5)$.
$T_3 < T_8$ because $(b, a, 6) < (b, a, 7)$.

Induced Sorting

Define three type of suffixes $-, +$ and $\ast$ as follows:
\[
C_- = \{i \in [0..n) | T[i] > T[i+1] \}
\]
\[
C_+ = \{i \in [0..n) | T[i] < T[i+1] \}
\]
\[
C^\ast = \{i \in C^+ \mid i-1 \in C^- \}
\]

Example 4.29:
\[
\begin{array}{c}
0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12
T[i] = a \ b \ a \ d \ a \ b \ b \ a \ d \ o \\
N_i = 1 \ 4 \ 1 \ 2 \ 6 \ 5 \ 3 \ 1 \ 7 \ 8 \ 1
\end{array}
\]
For any $a \in \Sigma$ and $x \in \{\pm, +\}$ define:
\[
C_a = \{i \in [0..n) | T[i] = a \}
\]
\[
C_a = C_a \cap C^\ast
\]
and thus, if $i \in C_{C_2}$ and $j \in C^\ast_{C_2}$, then $T_i < T_j$. Hence the suffix array is
$aC_1C_2C_3..C_n = aC_1C_1^\ast C_2^\ast C_3^\ast ..C_n^\ast C_1C_2$.

To induce $C^\ast$-suffixes:
1. Set $C_a^\ast$ empty for all $a \in [1..\sigma]$.
2. For all suffixes $T_i$ such that $i-1 \in C^\ast$ 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 $i \in nC_1C_2C_3..C_n$.

Algorithm 4.31: InduceMinusSuffixes

Input: Lexicographically sorted lists $C_a$, $a \in \Sigma$
Output: Lexicographically sorted lists $C_a^\ast$, $a \in \Sigma$

(1) for $a \in \Sigma$ do $C_a^\ast \leftarrow \emptyset$
(2) pushback($a \leftarrow 1, C_{T[1]}^\ast$)
(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($i-1, C_{T[i-1]}^\ast$)
(6) for $i \in C_a^\ast$ do pushback($i-1, C_{T[i-1]}^\ast$)

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

The basic idea of induced sorting is to use information about the order of $T$, to induce the order of the suffix $T_{i-1} = T[i - 1]$.

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 $q = \log_2 n$, the time complexity is $O(n \log n)$ and the space needed in addition to the text and the suffix array is $O(n/\sqrt{\log n})$.

Example 4.32: InducePlusSuffixes

Input: Lexicographically sorted lists $C_a^\ast$, $a \in \Sigma$
Output: Lexicographically sorted lists $C_a^\ast$, $a \in \Sigma$

(1) for $a \in \Sigma$ do $C_a^\ast \leftarrow \emptyset$
(2) for $a \leftarrow \sigma - 1$ downto 1 do
(3) for $i \in C_a^\ast$ in reverse order do // include elements added during loop
(4) if $i > 0$ and $T[i-1] \leq a$ then pushfront($i-1, C_{T[i-1]}^\ast$)
(5) for $i \in C_a^\ast$ in reverse order do
(6) if $i > 0$ and $T[i-1] < a$ then pushfront($i-1, C_{T[i-1]}^\ast$)

Inducing $\ast$-type suffixes goes similarly but in reverse order so that again $i$ is always inserted before $i-1$:
1. Set $C_a^\ast$ empty for all $a \in [1..\sigma]$.
2. For all suffixes $T_i$ such that $i-1 \in C^\ast$ in descending lexicographical order, append $i-1$ into $C_{T[i-1]}^\ast$.

Algorithm 4.33: SAI5

Step 0: Choose $C$.

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

Example 4.35:
\[
\begin{array}{c}
0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12 \ 13 \ 14
T[i] = a \ b \ a \ d \ a \ b \ b \ a \ d \ o \\
N_i = 1 \ 4 \ 1 \ 2 \ 6 \ 5 \ 3 \ 1 \ 7 \ 8 \ 1
\end{array}
\]

$C_1 = (2, 5, 8)$, $C_2 = C_3^\ast = C_4^\ast = \emptyset$, $C = (2, 5, 8, 14)$. 

Note that $\ell$ is in $C^\ast$ and thus $\ell - 1 \in C^-$. By Lemma 4.30, $T[\ell] < T[\ell - 1]$.
$T[\ell - 1, A] = T[\ell - 1, 1]$ and thus $T[\ell] < T[\ell - 1]$. If we had $\ell \in C^+$, we would have $\ell \in C^-$. Since this is not the case, we must have $\ell \in C^\ast$.
$T[\ell] < T[\ell - 1]$. Since $T[\ell, k] = T[\ell, \ell]$, $T_i < T_{\overline{i}}$, $T_i < T_{\overline{i}}$. 

**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 order preserving names $N_i \in [1, |C|] - 1$ to the string $S'_i$ so that $N_i < N_j$ iff $S'_i < S'_j$.
- Construct the sequence $R = N_iN_iN_i...N_i0$, where $i_1 < i_2 < \cdots < i_6$ 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:**

$$
\begin{array}{cccccccccccc}
T[i] & n & i & s & s & i & s & s & i & p & i & p & i & i & s \\
N_i & 2 & 2 & 2 & 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
R &= [i] [s] [i] [s] [i] [s] [i] [p] [i] [p] [i] [i] [s] \Rightarrow 2210, \ SA_R = (3, 2, 1, 0), C'_1 = (8, 5, 2)
\end{array}
$$

**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 $C'_1C'_2...C'_{\sigma-1}$.

With this change, most of the work is done in the induction procedures. This is very fast in practice, because all the lists $C'_a$ 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.

**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 $SA = nC'_1 C'_2 C'_3 \ldots C'_{\sigma-1} C'_\sigma$.

**Example 4.37:**

<table>
<thead>
<tr>
<th>$T[i]$</th>
<th>n i s s</th>
<th>s i</th>
<th>s i</th>
<th>s i</th>
<th>p p</th>
<th>i i</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13 14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n = 14$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C'_1 = (13, 12)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C'_2 = (13, 12, 8, 5, 2)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C'_3 = (14, 13, 12, 8, 9, 5, 2, 1, 0, 11, 10, 7, 4, 6, 3)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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, ...