Algorithm 4.15: Backward Search
Input: array C, function rankL, pattern P
Output: suffix array range [b..e) containing starting positions of P
(1) \( b \leftarrow 0; \ c \leftarrow n + 1 \)
(2) for \( i \leftarrow m - 1 \) downto 0 do
(3) \( c \leftarrow P[i] \)
(4) \( b \leftarrow C[c] + \text{rank}_L(c, b) \)
(5) \( e \leftarrow C[c] + \text{rank}_L(c, e) \)
(6) return \( [b..e) \)
• The array \( C \) requires an integer alphabet that is not too large.
• The trivial implementation of the function rankL, 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 an important component in many compressed text indexes. These are covered in the course Data Compression Techniques.

Suffix Array Construction
Suffix array construction means simply sorting the set of all suffixes.
• Using standard sorting or string sorting the time complexity is \( \Omega(S \cdot \text{LCP}(T_{[0..n]})) \).
• 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 \( O(n) \) on a constant alphabet.

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

LCP Array Construction
The LCP array is easy to compute in linear time from the suffix array with the help of a couple of additional arrays:
• For each \( i \in [1..n] \), let \( \Phi[S|a|i|] = S|a|i-1 \). Then the suffix \( T_{\Phi[i]} \) is the immediate lexicographical predecessor of the suffix \( T_i \).
• For each \( i \in [1..n] \), let \( \text{PLCP}[S|a|i|] = LCP[P_i] \). Then \( \text{PLCP}[i] = LCP[S|a|i-1|] = \text{lcpc}(T_{\Phi[i]}, T_i) \), i.e., \( \text{PLCP}[i] \) is the lcp between \( T_i \) and its lexicographical predecessor.

Example 4.16: \( \text{PLCP}[S|a|] \)
\[
\begin{array}{cccccccc}
 i & S|a|i & \text{LCP}[T_{SA|i}] & j & S|a|i-1 & \Phi[j] & \text{PLCP}[j] & T_j \\
 0 & b & $ & 0 & 0 & 1 & u & \text{banana} \\
 1 & a & a$ & 2 & 2 & 3 & 3 & \text{ana}$ \\
 2 & a & a$ & 2 & 2 & 3 & 3 & \text{ana}$ \\
 3 & a & a$ & 2 & 2 & 3 & 3 & \text{ana}$ \\
 4 & a & a$ & 2 & 2 & 3 & 3 & \text{ana}$ \\
 5 & a & a$ & 2 & 2 & 3 & 3 & \text{ana}$ \\
 6 & a & a$ & 2 & 2 & 3 & 3 & \text{ana}$ \\
\end{array}
\]

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

Algorithm 4.18: LCP array construction
Input: text \( \mathbf{T}_{[0..n]} \), suffix array \( S|a| \), inverse suffix array \( S|a|-1| \)
Output: LCP array \( \text{LCP}[T_{[0..n]}] \)
(1) for \( i \in [1..n] \) do \( \Phi[S|a|i|] \leftarrow S|a|i-1 \)
(2) \( \ell \leftarrow 0 \)
(3) for \( j \leftarrow 0 \) to \( n-1 \) do
(4) \( T[j+\ell] = \Phi[S|a|i|] \leftarrow \ell \leftarrow \ell + 1 \)
(5) \( \text{PLCP}[i] \leftarrow \ell \)
(6) \( \ell > 0 \) then \( \ell \leftarrow \ell - 1 \)
(7) for \( i \in [1..n] \) do \( \text{LCP}[i] \leftarrow \text{PLCP}[S|a|i|] \)
(8) return \( \text{LCP} \)

The time complexity is \( O(n) \):
• Everything except the while loop on line (4) takes clearly linear time.
• Each round in the loop increments \( \ell \). Since \( \ell \) 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 \( T_i^j \) denote the text factor \( T[i..j] \). \( \min(i + \ell, n + 1) \) and call it an \( \ell \)-factor. In other words:
• \( T_i^j \) is the factor starting at \( i \) and of length \( \ell \) except when the factor is cut short by the end of the text.
• \( T_i^j \) is the prefix of the suffix \( T_i \) of length \( \ell \), or \( T_i \) when \( |T_i| < \ell \).

The idea is to sort the sets \( T_i^j \) for ever increasing values of \( \ell \).
• First sort \( T_i^0 \), which is equivalent to sorting individual characters. This can be done in \( O(n \log n) \) time.
• Then, for \( \ell = 1, 2, 4, 8, \ldots \), use the sorted set \( T_i^j \) to sort the set \( T_{i+2}^j \) in \( O(n) \) time.
• After \( O(n \log n) \) rounds, \( \ell > n \) and \( T_i^j \) is sorted, so we have sorted the set of all suffixes.

Now, given \( N_i^j \), for the purpose of sorting, we can use
• \( N_i^j \) to represent \( T_i^j \)
• the pair \( (N_i, N_{i+2}) \) to represent \( T_{i+2}^j = T_i^j T_{i+2}^j \).

Thus we can sort \( T_{i+2}^j \) by sorting pairs of integers, which can be done in \( O(n) \) time using LSD radix sort.

Theorem 4.20: The suffix array of a string \( \mathbf{T}_{[0..n]} \) 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.
Let us return to the first phase of the prefix doubling algorithm: assigning names $N^1_i$ 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 a general alphabet, we can use ternary quicksort for time complexity $O(n \log_3 n)$, where $n_3$ is the number of distinct symbols in $T$.
- On an integer alphabet of size $n^i$ for any constant $i$, we can use LSD radix sort with radix $n$ for time complexity $O(n)$.

After this, we can replace each character $T[i]$ with $N^1_i$ 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., $\perp$.
- 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.

**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 $T[n] = \perp$ (i.e., $T'[n] = 0$ 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 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.
3. Sort the set $T[R,\perp]$ 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:** $D_q = \{1, 2, 4\}$

$$1 - 1 = 0 \quad 1 - 4 = -3 = 4 \mod q$$

$$2 - 1 = 1 \quad 2 - 4 = -2 = 5 \mod q$$

$$4 - 2 = 2 \quad 1 - 2 = -1 = 6 \mod q$$

$$4 - 1 = 3$$

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

**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 $C = C_1 \cup C_2$ and $\bar{C} = C_0$.

**Example 4.25:**

\begin{align*}
T[i] & \quad y \ a \ b \ b \ a \ d \ a \ b \ b \ a \ d \ o \$ \\
\bar{C} = C_0 = [0, 3, 6, 9, 12], \quad C_1 = [1, 4, 7, 10], \quad C_2 = [2, 5, 8, 11] \text{ and } C = [1, 2, 4, 5, 7, 8, 10, 11].
\end{align*}

**Step 1:** Sort $T_C$.

- Replace each factor $T^3_r$ in $R$ with an order preserving name $N^3_j \in [1..|R|]$. The names can be computed by sorting the characters with LSD radix sort in $O(n)$ time. Let $R'$ be the result with $0 \mod q$.
- Construct the inverse suffix array $SA^3_j$ of $R'$. This is done recursively using DC3 unless all symbols in $R'$ are unique, in which case $SA^3_j = R'$.
- From $SA^3_j$, we get order preserving names for suffixes in $T_C$.
- For $i \in C$, let $N_i = SA^3_j[j]$, where $j$ is the position of $T^3_r$ in $R'$.
- For $i \in \bar{C}$, let $N_i = \perp$. Also let $N_{n+1} = N_{n+2} = 0$.

**Example 4.26:**

\begin{align*}
\text{Let } R' \text{ be a reduced string of length } |C|, \text{ and } T'[i] \text{ be the } i\text{-th character of } T'. \\
T[i] & \quad y \ a \ b \ b \ a \ d \ a \ b \ b \ a \ d \ o \$ \\
N_i & \quad \perp \perp 1 1 4 4 7 7 4 4 6 3 8 0 \\
\end{align*}

**Step 2:**

- **Step 2(a):** Sort $T_C$.
  - For each $i \in C$, we represent $T[i]$ with the pair $(T[i], N_{i+1})$. Then $T_i \leq T_{i'} \iff (T[i], N_{i+1}) \leq (T[i'], N_{i'+1})$.
  - Note that $N_{n+1} \perp$ 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:**

\begin{align*}
T[i] & \quad y \ a \ b \ b \ a \ d \ a \ b \ b \ a \ d \ o \$ \\
N_i & \quad \perp 1 1 4 4 7 7 4 4 6 3 8 0 \\
\end{align*}

$$T_{12} < T_6 < T_5 < T_3 < T_2 \text{ because } 10 < 9 \leq 7 \leq 8 \leq 1.$$
Step 2(b): Merge $T_C$ and $\overline{T_C}$.

- Use comparison based merging algorithm needing $O(n)$ comparisons.
- To compare $T_i \in T_C$ and $T_j \in \overline{T_C}$, we have two cases:
  \( i \in C_i \land T_i \leq T_{i+1} \iff (T[i], N_{i+1}) \leq (T[i], N_{i+1}) \)
  \( i \in C_i \land T_i \leq T_{i+1} \iff (T[i], N_{i+1} + 1, N_{i+2}) \leq (T[i], T[i+1], N_{i+2}) \)

  Note that none of the $N$-values is $i$.

Example 4.28:

\[
\begin{array}{cccccccccccccccc}
\text{Ti} & \text{Ni} & \text{i} & \text{1} & \text{2} & \text{3} & \text{4} & \text{5} & \text{6} & \text{7} & \text{8} & \text{9} & \text{10} & \text{11} & \text{12} \\
1 & 4 & 1 & 4 & 1 & 4 & 1 & 4 & 1 & 4 & 1 & 4 & 1 & 4 \\
1 & 4 & 2 & 6 & 5 & 3 & 7 & 8 & 1 & 4 & 1 & 4 & 1 & 4 \\
\end{array}
\]

$T_1 < T_6$ 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 $C_i^-$, $a \in \{1..\sigma\}$.
2. Use $C_i^+$, $a \in \{1..\sigma\}$, to induce the order of the sets $C_i^-$, $a \in \{1..\sigma\}$.
3. Use $C_i^+$, $a \in \{1..\sigma\}$, to induce the order of the sets $C_i^+$, $a \in \{1..\sigma\}$.

The suffixes involved in the induction steps can be identified using the following rules (proof is left as an exercise).

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

**Induced Sorting**

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

- $C_i^-$ = \{ $i \in \{0..n\}$ | $T_i > T_{i+1}$ \}
- $C_i^+$ = \{ $i \in \{0..n\}$ | $T_i < T_{i+1}$ \}
- $C_i^*$ = \{ $i \in C_i^+$ | $i - 1 \notin C_i^-$ \}

Example 4.30:

\[
\begin{array}{cccccccccccccccc}
\text{Ti} & \text{Ni} & \text{i} & \text{1} & \text{2} & \text{3} & \text{4} & \text{5} & \text{6} & \text{7} & \text{8} & \text{9} & \text{10} & \text{11} & \text{12} \\
1 & 4 & 1 & 4 & 1 & 4 & 1 & 4 & 1 & 4 & 1 & 4 & 1 & 4 \\
1 & 4 & 2 & 6 & 5 & 3 & 7 & 8 & 1 & 4 & 1 & 4 & 1 & 4 \\
\end{array}
\]

For every $a \in \Sigma$ and $x \in (\ldots, +, -)$ define

- $C_i^a$ = \{ $i \in \{0..n\}$ | $T[i] = a$ \}
- $C_i^a$ = $\emptyset$
- $C_i^a$ = \{ $i \in C_i^a$ | $T[i] > a$ \}

and thus, if $i \in C_i^a$ and $j \in C_i^b$, then $T_i < T_j$. Hence the suffix array is $nC_1C_2...C_{\sigma-1} = nC_1C_{\sigma-1}C_{\sigma-2}...C_2C_1$.

To induce $C_i^-$ suffixes:

1. Set $C_i^a$ empty for all $a \in \{1..\sigma\}$.
2. For all suffixes $T_i$ such that $i - 1 \in C_i^-$ in lexicographical order, append $i - 1$ into $C_{T[i-1]}$.

By Lemma 4.25(a), Step 2 can be done by checking the relevant conditions for all $i \in nC_i^1C_i^2...C_{\sigma-1}$.

**Algorithm 4.32: InduceMinusSuffixes**

**Input:** Lexicographically sorted lists $C_i^a$, $a \in \Sigma$

**Output:** Lexicographically sorted lists $C_i^a$, $a \in \Sigma$

1. for $a \in \Sigma$ do $C_i^a = \emptyset$
2. pushback($n - 1, C_{T[n-1]}$)
3. for $a \in \sigma$ downto 1 do
4. for $i \in C_i^a$ do $f / /$ 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 $i \in C_i^a$ do pushback($i - 1, C_{T[i-1]}$)

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

We still need to explain how to sort the $*$-type suffixes. Define

\[ F[i] = \min\{k \in \{i + 1..n\} | k \in C_i^* \lor k = n\} \]
\[ S_i^*= T[i, F[i]] \]
\[ S_i^+ = S_i^* \]

where $e$ is a special symbol larger than any other symbol.

**Lemma 4.34:** For any $i, j \in \{0..n\}$, $T_i < T_j$ iff $S_i^* < S_j^*$ or $S_i^* = S_j^*$ and $T_{i+j} < T_{i+j}$.

**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 $T_{i+j} < T_{i+j}$ by the claim. We will show that this is correct.

Let $e = F[i]$ and $k = i + e - 1$. Then

- $e \in C_i^*$ and thus $i - 1 \in C_i^-$. By Lemma 4.25(b), $T[i-1] > T[e]$.
- $T[k - 1, a] = T[y - 1, a]$ and thus $T[k - 1] > T[y]$. If we had $k \in C_i^+$, we would have $k \in C_i^-$. Since this is not the case, we must have $k \in C_i^+$.

Note that $T[e]$. Since $e \in C_i^*$ and $k \in C_i^+$, we must have $T_e < a^\infty < T_i$. Since $T[i, k] = T[j, l]$ and $T_i < T_j$, we have $T_i < T_j$.

**Algorithm 4.35: SAIS**

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

- Compute the types of suffixes. This can be done in $O(n)$ time based on Lemma 4.25.

  Set $C = U_{a\in\{1..\sigma\}} C_{i+a}^i \{n\}$. Note that $|C| \leq n/2$, since for all $i \in C$, $i - 1 \in C^- \subseteq C$.

**Example 4.36:**

\[
\begin{array}{cccccccccccccccc}
\text{Ti} & \text{Ni} & \text{i} & \text{1} & \text{2} & \text{3} & \text{4} & \text{5} & \text{6} & \text{7} & \text{8} & \text{9} & \text{10} & \text{11} & \text{12} & \text{13} & \text{14} \\
1 & 4 & 1 & 4 & 1 & 4 & 1 & 4 & 1 & 4 & 1 & 4 & 1 & 4 & 1 & 4 \\
1 & 4 & 2 & 6 & 5 & 3 & 7 & 8 & 1 & 4 & 1 & 4 & 1 & 4 & 1 & 4 \\
\end{array}
\]
Step 1: Sort $T[0..n]$.

- 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 $N_i \leq N_j$ iff $S_i \leq S_j$.
- Construct the sequence $R = N_1 N_2 \ldots N_0$, where $i_1 < i_2 < \ldots < i_k$ 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 4.37:

$$
\begin{align*}
& i \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \\
& T[i] \quad n \quad n \quad n \quad i \quad s \quad i \quad s \quad i \quad s \quad i \quad p \quad p \quad i \quad i \quad $ \\
& N_i \quad 2 \quad 2 \quad 2 \quad 1 \quad 1 \\
& R = [issi][issi][iippii]$ = 2210, \quad SA_R = (3, 2, 1, 0), \quad C^*_1 = (8, 5, 2)
\end{align*}
$$

Theorem 4.39: 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 $C^+_1 C^+_2 \ldots C^+_k$, $a \in [1..\sigma]$.

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 $SA = nC^*_1 C^*_2 C^*_3 \ldots C^*_k$, $C^*_a = (\sigma)$.

Example 4.38:

$$
\begin{align*}
& T[i] \quad n \quad n \quad n \quad i \quad s \quad i \quad s \quad i \quad i \quad p \quad p \quad i \quad i \quad $ \\
& \text{type of } T_i \quad - \quad - \quad * \quad - \quad - \quad * \quad - \quad - \quad * \quad + \quad - \quad - \quad - \\
& n = 14 \quad \Rightarrow \quad C^*_1 = (13, 12) \\
& C^*_1 C^*_2 = (13, 12, 8, 5, 2) \quad \Rightarrow \quad C^*_a = (1, 0), \quad C^*_p = (11, 10), \quad C^*_p = (7, 4, 6, 3) \\
& \Rightarrow \quad C^*_1 = (8, 9, 5, 2) \\
& \Rightarrow \quad SA = C^*_1 C^*_2 C^*_3 C^*_4 = (14, 13, 12, 8, 9, 5, 2, 1, 0, 11, 10, 7, 4, 6, 3)
\end{align*}
$$

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, ...

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