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 $\mathcal{M}$ beginning with $P_i = P[i..m)$, 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.
- $rank_L(P[i], b_{i+1})$ 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 $P[i]P_{i+1} = P_i$.
- Thus $b_i = C[P[i]] + rank_L(P[i], b_{i+1})$.

Computing e_i from e_{i+1} is similar: $e_i = C[P[i]] + rank_L(P[i], e_{i+1})$.

Algorithm 4.15: Backward Search
Input: array C, function $rank_L$, pattern P
Output: suffix array range $[b..e)$ containg starting positions of P

```
b ← 0; e ← n + 1
for i ← m − 1 downto 0 do
    c ← P[i]
    b ← C[c] + rank_L(c, b)
    e ← C[c] + rank_L(c, e)
return [b..e)
```

- The array C requires an integer alphabet that is not too large.
- The trivial implementation of the function $rank_L$ as an array requires $\Theta(\sigma n)$ 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 additional arrays:

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

Example 4.16: $T =$ banana$.

i	$SA[i]$	$LCP[i]$	$T_{SA[i]}$	j	$SA^{-1}[j]$	$\Phi[j]$	$PLCP[j]$	T_j
0	6		$	0	4	1	0	banana$
1	5	0	a$	1	3	3	3	anana$
2	3	1	ana$	2	6	4	2	nana$
3	1	3	anana$	3	2	5	1	ana$
4	0	0	banana$	4	5	0	0	na$
5	4	0	na$	5	1	6	0	a$
6	2	2	nana$	6	0			$

The idea is to compute the lcp values by comparing the suffixes, but skip a prefix based on a known lower bound for the lcp value obtained using the following result.

Lemma 4.17: For any $j \in [1..n)$, $PLCP[j] \geq PLCP[j-1] - 1$

Proof.

- Let $\ell = PLCP[j-1]$ and $\ell' = LCP[j]$. We want to show that $\ell' \geq \ell - 1$. If $\ell = 0$, the claim is trivially true.
- If $\ell > 0$, then for some symbol c, $T_{j-1} = cT_j$ and $T_{\Phi[j-1]} = cT_{\Phi[j-1]+1}$. Thus $T_{\Phi[j-1]+1} < T_j$ and $lcp(T_j, T_{\Phi[j-1]+1}) = lcp(T_{j-1}, T_{\Phi[j-1]}) - 1 = \ell - 1$.
- If $\Phi[j] = \Phi[j-1] + 1$, then $\ell' = lcp(T_j, T_{\Phi[j]}) = lcp(T_j, T_{\Phi[j-1]+1}) = \ell - 1$.
- If $\Phi[j] \neq \Phi[j-1] + 1$, then $T_{\Phi[j-1]+1} < T_{\Phi[j]} < T_j$ because $T_{\Phi[j]}$ is the *immediate* lexicographical predecessor of T_j. Thus $\ell' = lcp(T_j, T_{\Phi[j]}) \geq lcp(T_j, T_{\Phi[j-1]+1}) = \ell - 1$.

□

The algorithm computes first Φ then $PLCP$ and finally LCP. The computation of $PLCP$ 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 ∈ [1..n] do Φ[SA[i]] ← SA[i − 1]
(2) ℓ ← 0
(3) for j ← 0 to n − 1 do
(4)     while T[j + ℓ] = T[Φ[j] + ℓ] do ℓ ← ℓ + 1
(5)     PLCP[j] ← ℓ
(6)     if ℓ > 0 then ℓ ← ℓ − 1
(7) for i ∈ [1..n] do LCP[i] ← PLCP[SA[i]]
(8) return LCP
```

The time complexity is $\mathcal{O}(n)$ 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 $\mathcal{O}(n)$ times in total.

Suffix Array Construction

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

- Using standard sorting or string sorting the time complexity is $\Omega(\Sigma LCP(T_{[0..n]}))$.
- 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 $\mathcal{O}(n)$ in the constant alphabet model.

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

Prefix Doubling

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

Let T_i^ℓ denote the text factor $T[i..\min\{i+\ell, n+1\})$ and call it an ℓ-factor. In other words:

- T_i^ℓ 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^ℓ is the prefix of the suffix T_i of length ℓ, or T_i when $|T_i| < \ell$.

The idea is to sort the sets $T_{[0..n]}^\ell$ for ever increasing values of ℓ.

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

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

$$N_i^\ell \leq N_j^\ell \text{ if and only if } T_i^\ell \leq T_j^\ell .$$

Then, for $\ell > n$, $N_i^\ell = SA^{-1}[i]$.

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

$$N_i^\ell = |\{j \in [0, n] \mid T_j^\ell < T_i^\ell\}| .$$

Example 4.19: Prefix doubling for $T =$ banana$.

N^1		N^2		N^4		$N^8 = SA^{-1}$	
4	b	4	ba	4	bana	4	banana$
1	a	2	an	3	anan	3	anana$
5	n	5	na	6	nana	6	nana$
1	a	2	an	2	ana$	2	ana$
5	n	5	na	5	na$	5	na$
1	a	1	a$	1	a$	1	a$
0	$	0	$	0	$	0	$

Now, given N^ℓ, for the purpose of sorting, we can use

- N_i^ℓ to represent T_i^ℓ
- the pair $(N_i^\ell, N_{i+\ell}^\ell)$ to represent $T_i^{2\ell} = T_i^\ell T_{i+\ell}^\ell$.

Thus we can sort $T_{[0..n]}^{2\ell}$ by sorting pairs of integers, which can be done in $\mathcal{O}(n)$ time using LSD radix sort.

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

Let us return to the first phase of the prefix doubling algorithm: assigning names N_i^1 to individual characters. This is done by sorting the characters, which is easily within the time bound $\mathcal{O}(n \log n)$, but sometimes we can do it faster:

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

After this, we can replace each character $T[i]$ with N_i^1 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.

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 $T[n] = 0$ (=$ 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.

Assume that

- $|C| \leq \alpha n$ 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 $t(n) = \mathcal{O}(n) + t(\alpha n)$, whose solution is $t(n) = \mathcal{O}(n)$.

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 cover sampling
- SAIS uses induced sorting

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:

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

Example 4.21: $D_7 = \{1, 2, 4\}$

$$\begin{array}{ll} 1-1=0 & 1-4=-3\equiv 4 \pmod q \\ 2-1=1 & 2-4=-2\equiv 5 \pmod q \\ 4-2=2 & 1-2=-1\equiv 6 \pmod q \\ 4-1=3 & \end{array}$$

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

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

Example 4.22: If $T =$ banana$ and $D_3 = \{1, 2\}$, then $C = \{1, 2, 4, 5\}$ and $T_C = \{$anana$, nana$, na$, a$$\}$.

Once we have sorted the difference cover sample T_C, we can compare any two suffixes in $\mathcal{O}(q)$ 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 ℓ 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 4.23: $D_3 = \{1, 2\}$ and $C = \{1, 2, 4, 5, \ldots\}$

$$\begin{array}{lll} T_0 = T[0]T_1 & T_0 = T[0]T[1]T_2 & T_0 = T[0]T_1 \\ T_1 = T[1]T_2 & T_2 = T[2]T[3]T_4 & T_3 = T[3]T_4 \end{array}$$

Algorithm 4.24: DC3

Step 0: Choose C.

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

Example 4.25:

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	$

$\bar{C} = C_0 = \{0, 3, 6, 9, 12\}$, $C_1 = \{1, 4, 7, 10\}$, $C_2 = \{2, 5, 8, 11\}$ and $C = \{1, 2, 4, 5, 7, 8, 10, 11\}$.

Step 1: Sort T_C.

- For $k \in \{1, 2\}$, Construct the strings $R_k = (T_k^3, T_{k+3}^3, T_{k+6}^3, \ldots, T_{\max C_k}^3)$ whose characters are 3-factors of the text, and let $R = R_1 R_2$.
- Replace each factor T_i^3 in R with an order preserving name $N_i^3 \in [1..|R|]$. The names can be computed by sorting the factors with LSD radix sort in $\mathcal{O}(n)$ time. Let R' be the result appended with 0.
- Construct the inverse suffix array $SA_{R'}^{-1}$ of R'. This is done recursively using DC3 unless all symbols in R' are unique, in which case $SA_{R'}^{-1} = R'$.
- From $SA_{R'}^{-1}$, we get order preserving names for suffixes in T_C. For $i \in C$, let $N_i = SA_{R'}^{-1}[j]$, where j is the position of T_i^3 in R. For $i \in \bar{C}$, let $N_i = \perp$. Also let $N_{n+1} = N_{n+2} = 0$.

Example 4.26:

R	abb	ada	bba	do$	bba	dab	bad	o$	
R'	1	2	4	7	4	6	3	8	0
$SA_{R'}^{-1}$	1	2	5	7	4	6	3	8	0

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
  $$T_i \le T_j \iff (T[i], N_{i+1}) \le (T[j], N_{j+1}) \ .$$
  Note that $N_{i+1} \ne \bot$ 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 4.27:

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	$\bot$

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

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

- Use comparison based merging algorithm needing $\mathcal{O}(n)$ comparisons.
- To compare $T_i \in T_C$ and $T_j \in T_{\bar{C}}$, we have two cases:
  $$i \in C_1 : T_i \le T_j \iff (T[i], N_{i+1}) \le (T[j], N_{j+1})$$
  $$i \in C_2 : T_i \le T_j \iff (T[i], T[i+1], N_{i+2}) \le (T[j], T[j+1], N_{j+2})$$
  Note that none of the N-values is $\bot$.

Example 4.28:

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	$\bot$

$T_1 < T_6$ because $(\mathtt{a}, 4) < (\mathtt{a}, 5)$.
$T_3 < T_8$ because $(\mathtt{b}, \mathtt{a}, 6) < (\mathtt{b}, \mathtt{a}, 7)$.

Theorem 4.29: Algorithm DC3 constructs the suffix array of a string $T[0..n)$ in $\mathcal{O}(n)$ 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 $\mathcal{O}(n \log n)$ and the space needed in addition to the text and the suffix array is $\mathcal{O}(n/\sqrt{\log n})$.

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:

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

Example 4.30:

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	\$
type of T_i	$-$	$+$	$-$	$-$	$+$	$-$	$+$	$-$	$-$	$+$	$+$	$-$	
		$*$	$*$		$*$	$*$	$*$	$*$		$*$		$*$	

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

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

The two types of suffixes starting with the same character are lexicographically separated:

Lemma 4.31: For all $a \in \Sigma$,

$$C_a^- = \{i \in C_a \mid T_i < a^\infty\}$$
$$C_a^+ = \{i \in C_a \mid T_i > a^\infty\}$$

Thus, if $i \in C_a^-$ and $j \in C_a^+$, then $T_i < T_j$. Hence the suffix array is $nC_1C_2 \dots C_{\sigma-1} = nC_1^-C_1^+C_2^-C_2^+ \dots C_{\sigma-1}^-C_{\sigma-1}^+$.

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 $a \in [1..\sigma)$

(a) $i - 1 \in C_a^-$ iff $i > 0$ and $T[i-1] = a$ and one of the following holds
   1. $i = n$
   2. $i \in C^{+*}$
   3. $i \in C^-$ and $T[i-1] \ge T[i]$.

(b) $i - 1 \in C_a^+$ iff $i > 0$ and $T[i-1] = a$ and one of the following holds
   1. $i \in C^{-*}$
   2. $i \in C^+$ and $T[i-1] \le T[i]$.

To induce C^- 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 4.32(a), Step 2 can be done by checking the relevant conditions for all $i \in nC_1^-C_1^{+*}C_2^-C_2^{+*}\dots$.

Algorithm 4.33: InduceMinusSuffixes
Input: Lexicographically sorted lists C_a^{+*}, $a \in \Sigma$
Output: Lexicographically sorted lists C_a^-, $a \in \Sigma$

```
for $a \in \Sigma$ do $C_a^- \leftarrow \emptyset$
pushback($n-1, C_{T[n-1]}^-$)
for $a \leftarrow 1$ to $\sigma - 1$ do
    for $i \in C_a^-$ do    // include elements added during the loop
        if $i > 0$ and $T[i-1] \ge a$ then pushback($i-1, C_{T[i-1]}^-$)
    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 $i-1$.

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 4.34: InducePlusSuffixes
Input: Lexicographically sorted lists C_a^{-*}, $a \in \Sigma$
Output: Lexicographically sorted lists C_a^+, $a \in \Sigma$

```
for $a \in \Sigma$ do $C_a^+ \leftarrow \emptyset$
for $a \leftarrow \sigma - 1$ downto 1 do
    for $i \in C_a^+$ in reverse order do // include elements added during loop
        if $i > 0$ and $T[i-1] \le a$ then pushfront($i-1, C_{T[i-1]}^+$)
    for $i \in C_a^{-*}$ do pushfront($i-1, C_{T[i-1]}^+$)
```

We still need to explain how to sort the $-*$-type suffixes. For this we need the following definition and result:

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

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 $T_{F[i]} < T_{F[j]}$.

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

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

□

Algorithm 4.36: SAIS

Step 0: Choose C.

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

Example 4.37:

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	$
type of T_i	$-$	$+$	$-$	$-$	$+$	$-$	$+$	$-$	$-$	$+$	$+$	$-$	
		*	*		*	*	*	*		*		*	

$C_{\mathtt{b}}^{-*} = \{2, 7\}$, $C_{\mathtt{d}}^{-*} = \{5\}$, $C_{\mathtt{o}}^{-*} = \{11\}$, $C = \{2, 5, 7, 11\}$.

Step 1: Sort T_C.

- Sort the strings S_i, $i \in C^{-*}$. Since the total length of the strings S_i is $\mathcal{O}(n)$, the sorting can be done in $\mathcal{O}(n)$ time using LSD radix sort.
- Assign order preserving names $N_i \in [1..|C|]$ to the string S_i so that $N_i \leq N_j$ iff $S_i \leq S_j$.
- Construct the sequence $R = N_{i_1} N_{i_2} \dots N_k 0$, where $i_1 < i_3 < \cdots < i_k$ 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 4.38:

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			1			3		2				4	

$R = [\mathtt{bbad}][\mathtt{dab}][\mathtt{bbado}][\mathtt{o\$}]\mathtt{\$} = 13240$, $SA_R = (4, 0, 2, 1, 3)$, $C_{\mathtt{b}}^{-*} = (2, 7)$

Step 2: Sort $T_{[0..n]}$.

- 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 $SA = nC_1^{-}C_1^{+}C_2^{-}C_2^{+} \dots C_{\sigma-1}^{-}C_{\sigma-1}^{+}$.

Example 4.39:

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	$
type of T_i	$-$	$+$	$-$	$-$	$+$	$-$	$+$	$-$	$-$	$+$	$+$	$-$	

$C_{\mathtt{y}}^{+} = C_{\mathtt{y}}^{-*} = C_{\mathtt{o}}^{+} = \emptyset$, $C_{\mathtt{o}}^{-*} = \{11\} \quad \Rightarrow \quad C_{\mathtt{d}}^{+} = \{10\}$,
$C_{\mathtt{d}}^{+} = \{10\} \quad \Rightarrow \quad C_{\mathtt{a}}^{+} = \{9\}$, $C_{\mathtt{d}}^{-*} = \{5\} \quad \Rightarrow \quad C_{\mathtt{a}}^{+} = \{4, 9\}$, $C_{\mathtt{b}}^{+} = \emptyset$,
$C_{\mathtt{b}}^{-*} = \{2, 7\} \quad \Rightarrow \quad C_{\mathtt{a}}^{+}\{1, 6, 4, 9\}$, $C_{\mathtt{a}}^{+}\{1, 6, 4, 9\} \quad \Rightarrow \quad C_{\mathtt{a}}^{+*} = \{1, 6, 4, 9\}$
$n = 12 \quad \Rightarrow \quad C_{\mathtt{o}}^{-} = \{11\}$, $C_{\mathtt{a}}^{-} = \emptyset$,
$C_{\mathtt{a}}^{+*} = \{1, 6, 4, 9\} \quad \Rightarrow \quad C_{\mathtt{y}}^{-} = \{0\}, C_{\mathtt{d}}^{-} = \{5\}, C_{\mathtt{b}}^{-} = \{3, 8\}$,
$C_{\mathtt{b}}^{-} = \{3, 8\} \quad \Rightarrow \quad C_{\mathtt{b}}^{-} = \{3, 8, 2, 7\}$, $C_{\mathtt{b}}^{-} = \{\dots, 2, 7\}$, $C_{\mathtt{d}}^{-} = \{5\}$, $C_{\mathtt{d}}^{+*} = \emptyset$,
$C_{\mathtt{d}}^{-} = \{10\}$, $C_{\mathtt{o}}^{+*} = \emptyset$, $C_{\mathtt{y}}^{-} = \{0\}$, $C_{\mathtt{y}}^{+*} = \emptyset$
$SA = nC_{\mathtt{a}}^{-}C_{\mathtt{a}}^{+}C_{\mathtt{b}}^{-}C_{\mathtt{b}}^{+}C_{\mathtt{d}}^{-}C_{\mathtt{d}}^{+}C_{\mathtt{o}}^{-}C_{\mathtt{o}}^{+}C_{\mathtt{y}}^{-}C_{\mathtt{y}}^{+} = (12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0)$

Theorem 4.40: Algorithm SAIS constructs the suffix array of a string $T[0..n)$ in $\mathcal{O}(n)$ 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 InducePlusSuffixes to construct the lists C_a^{+}, $a \in [1..\sigma)$.
  3. Run InduceMinusSuffixes to construct the lists C_a^{-}, $a \in [1..\sigma)$.
  4. Remove non-*-type positions from $C_1^{-}C_2^{-} \dots C_{\sigma-1}^{-}$.

  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^{x} 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 $\mathcal{O}(n \log n)$ time and then continues as SAIS.

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

  Apostolico, Crochemore, Farach-Colton, Galil & Muthukrishnan: *40 years of suffix trees.* Communications of the ACM, 59(4), 2016, 66–73.

- Suffix tree construction

  Weiner: *Linear pattern matching algorithms.* Proc. 14th Annual IEEE Symposium on Switching and Automata Theory, IEEE 1973, 1–11.

  McCreight: *A space-economical suffix tree construction algorithm.* Journal of the ACM, 23(2), 1976, 262–272.

  Ukkonen: *On-line construction of suffix trees.* Algorithmica 14(3), 1995, 249–260.

  Farach: *Optimal suffix tree construction with large alphabets.* Proc. 38th IEEE Symposium on Foundations of Computer Science, IEEE 1997, 137–143.

- Suffix array

  Manber & Myers: *Suffix arrays: a new method for on-line string searches.* SIAM Journal on Computing, 22(5), 1993, 935–948.

- Enhanced suffix array

  Kasai, Lee, Arimura, Arikawa, Park: *Linear-time longest-common-prefix computation in suffix arrays and its applications.* Proc. 12th Symposium on Combinatorial Pattern Matching. LNCS 2089, Springer, 2001, 181–192.

  Abouelhoda, Kurtz & Ohlebusch: *Replacing suffix trees with enhanced suffix arrays.* Journal of Discrete Algorithms, 2(1), 2004, 53–86.

- Burrows–Wheeler transform

  Burrows & Wheeler: *A block sorting lossless data compression algorithm*. Technical Report 124, Digital Equipment Corporation, 1994.

- Backward search

  Ferragina & Manzini: *Indexing compressed text*. Journal of the ACM, 52(4), 2005, 552–581.

- LCP array construction

  Kasai, Lee, Arimura, Arikawa, Park: *Linear-time longest-common-prefix computation in suffix arrays and its applications*. Proc. 12th Symposium on Combinatorial Pattern Matching. LNCS 2089, Springer, 2001, 181–192.

  Kärkkäinen, Manzini &Puglisi: *Permuted longest-common-prefix array*. Proc. 20th Symposium on Combinatorial Pattern Matching. LNCS 5577, Springer, 2009, 181–192.

- Suffix array construction: survey

  Puglisi, Smyth & Turpin: *A taxonomy of suffix array construction algorithms*. ACM Computing Surveys, 39(2), Article 4, 2007.

- Suffix array construction: prefix doubling

  Manber & Myers: *Suffix arrays: a new method for on-line string searches*. SIAM Journal on Computing, 22(5), 1993, 935–948.

  Larsson & Sadakane: *Faster suffix sorting*. Theoretical Computer Science, 387(3), 2007, 258–272.

- Suffix array construction: difference cover sampling

  Kärkkäinen, Sanders & Burkhardt: *Linear work suffix array construction*. Journal of the ACM, 53(6), 2006, 918–936.

- Suffix array construction: induced sorting

  Ko & Aluru: *Space efficient linear time construction of suffix arrays*. Journal of Discrete Algorithms, 3(2), 2005, 143–156.

  Nong, Zhang & Chan: *Two efficient algorithms for linear time suffix array construction*. IEEE Transactions on Computers, 60(10), 2011, 1471–1484.