Now, given $N^i$, for the purpose of sorting, we can use
- $N^i_i$ to represent $T^i_i$
- the pair $(N^i_i, N^i_{i+\ell})$ to represent $T^i_i T^i_{i+\ell}$.

Thus we can sort $T^{[0..n]}_0$ by sorting pairs of integers, which can be done in $O(n \log n)$ time using LSD radix sort.

**Theorem 4.14:** 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.

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

After this, we can replace each character $T[i]$ with $N^i_i$ to obtain a new string $T^\prime$:
- The characters of $T^\prime$ are integers in the range $[0..n]$.
- The character $T^\prime[i]$ is 0 if $i$ is the unique, smallest symbol, i.e., $\$. 
- The suffix arrays of $T$ and $T^\prime$ are exactly the same.

Thus we can construct the suffix array using $T^\prime$ as the text instead of $T$. As we will see next, the suffix array of $T^\prime$ can be constructed in linear time. Then sorting the characters of $T$ to obtain $T^\prime$ is the asymptotically most expensive operation in the suffix array construction of $T$ for any alphabet.

**Algorithm 4.18: DC3**

**Step 0:** Choose $C$.
- Use difference cover $D_0$ of size $O(\sqrt{n})$.
- For any $q$, there exist a difference cover $D_q$ of size $O(\sqrt{n})$.
- The DC3 algorithm uses the simplest non-trivial difference cover $D_3 = \{1, 2\}$.

**Step 1:** Sort $T_C$.
- For $k \in \{1, 2\}$, construct the strings $R_k = (T^k_0 T^k_3 T^k_6 \ldots T^k_{\max})$, whose characters are 3-factors of the text, and let $R = R_1 R_2$.
- Replace each factor $T^k_\ell$ in $R$ with an order preserving name $N^k_\ell \in [1..|R|]$.
- 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 $SA_{R'}^{-1}$ of $R'$. This is done recursively using DC3 until 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 $k \in \{1, 2\}$, let $N_i = SA_{R'}^{-1}[1]$, where $i$ is the position of $T^k_i$ in $R$.
- For $k \in \{1, 2\}$, let $N_i = N_i R'$. Also let $N_{i+1} = N_i R' = 0$.

**Example 4.20:**
- Given sorted suffixes, where $C = \{i \in [0..n] | (i \mod q) \in D_q\}$.
- Once we have sorted the difference cover sample $T_C$, we can compare any two suffixes in $O(\sqrt{n})$ 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 $i$ such that $i + \ell \in C$ and $j + \ell \in C$. Where 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 $T[i + \ell]$ to $T[j + \ell]$, and if they are the same, then $T_{i+\ell}$ to $T_{j+\ell}$ using the sorted $T_C$.

**A difference cover sample** is a set $T_C$ of suffixes, where $C = \{i \in [0..n] | (i \mod q) \in D_q\}$.

**Example 4.16:** If $T = \text{banana}$ and $D_3 = \{1, 2\}$, then $C = \{1, 2, 4, 5\}$ and $T_C = \{\text{aanaa}, \text{anaaa}, \text{aaa}\}$.

**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 $t(n) = O(n) + t(\alpha n)$, 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 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 $T[0..n]$ is $[1..n]$ and that $T[n] = 0$ (as in the examples).

The outline of the algorithms is:
0. Choose a subset $C \subseteq [0..n]$.
1. Sort the set $T_C$.
   - (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:

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

**Example 4.15:** $D_q = \{1, 2, 4\}$.
- $1 - 1 = 0$ (mod $q$)
- $2 - 1 = 1$ (mod $q$)
- $2 - 2 = 0$ (mod $q$)
- $4 - 2 = 2$ (mod $q$)
- $4 - 4 = 0$ (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 $O(\sqrt{n})$.
- The DC3 algorithm uses the simplest non-trivial difference cover $D_3 = \{1, 2\}$.

**Algorithm 4.18: DC3**

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

**Example 4.19:** $T[i] = y a b b a d a b b a d$.

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

Let us return to the first phase of the prefix doubling algorithm: assigning names $N^i_i$ 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 $O(n \log n)$ where $\sigma_T$ is the number of distinct symbols in $T$.
- On an integer alphabet of size $n^\epsilon$ for any constant $\epsilon$, we can use LSD radix sort with radix $n$ for time complexity $O(n)$.

After this, we can replace each character $T[i]$ with $N^i_i$ to obtain a new string $T^\prime$:
- The characters of $T^\prime$ are integers in the range $[0..n]$.
- The character $T^\prime[i]$ is 0 if $i$ is the unique, smallest symbol, i.e., $\$. 
- The suffix arrays of $T$ and $T^\prime$ are exactly the same.

Thus we can construct the suffix array using $T^\prime$ as the text instead of $T$.
Step 2(a): Sort $T_C$.
- For each $i \in C$, we represent $T_i$ with the pair $(T[i], N_{i+1})$. Then $T_i \leq T_j \iff (T[i], N_{i+1}) \leq (T[j], N_{j+1})$.
- Note that $N_{i+1} \not= \bot$ for all $i \in C$.
- The pairs $(T[i], N_{i+1})$ are sorted by LSD radix sort in $O(n)$ time.

Example 4.21:


The suffixes involved in the induction steps can be identified using the following rules:

2. $T[i] < T[j]$ if $i > j$ and $j \in C^-$. Hence the suffix array is
3. $nC_{1C_{2} \ldots C_{σ−1}} = nC_{−} \ldots nC_{C_{1} \ldots C_{σ−1}}$.

Step 2(b): Merge $T_C$ and $T_C^-$.
- Use comparison based merging algorithm needing $O(n)$ comparisons.
- To compare $T_i \in T_C$ and $T_j \in T_C^-$, we have two cases:
  1. $i \in C_1$ and $T[i] \leq T[j] \iff (T[i], N_{i+1}) \leq (T[j], N_{j+1})$.
  2. $i \in C_2$ and $T[i] \leq T[j]$ if $i < j$ and $j \in C^-$. Hence the suffix array is

Theorem 4.23: 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. 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 4.24:

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

$$C_x = \{i \in [0..n) | i \in C^x\}$$

Then

$$C_+ = \{i \in \Sigma | T[i] < a\}$$

$$C_− = \{i \in \Sigma | T[i] > a\}$$

To induce $C$: 1. Set $C_x$ empty for all $a \in \{1..σ\}$.
2. For all suffixes $T_i$ such that $i \not= 1 \in C^+$ in lexicographical order, append $i$ into $C_+(T_i)$.

By Lemma 4.25(a), Step 2 can be done by checking the relevant conditions for all $i \in C_+(T_i..σ−1)$.

Algorithm 4.26: InduceMinusSuffixes
Input: Lexicographically sorted lists $C_x$, $a \in \Sigma$
Output: Lexicographically sorted lists $C_x^+$, $a \in \Sigma$

$C_x^+$ empty for all $a \in \{1..σ\}$.

For all suffixes $T_i$ such that $i \not= 1 \in C^+$ in descending lexicographical order, append $i$ into $C_+(T_i)$.

204

Induced Sorting
Define three type of suffixes $−$, $+$ and $∗$ as follows:

$$C− = \{i \in [0..n) | T[i] > T[i+1]\}$$

$$C+ = \{i \in [0..n) | T[i] < T[i+1]\}$$

$$C∗ = \{i \in C^∗ | i − 1 \in C−\}$$

To induce $C−$: 1. Set $C_x^−$ empty for all $a \in \{1..σ\}$.
2. For all suffixes $T_i$ such that $i − 1 \in C^−$ in lexicographical order, append $i$ into $C_−(T_i)$.

By Lemma 4.25(a), Step 2 can be done by checking the relevant conditions for all $i \in C_−(T_i..σ−1)$.

Algorithm 4.27: InducePlusSuffixes
Input: Lexicographically sorted lists $C_x^−$, $a \in \Sigma$
Output: Lexicographically sorted lists $C_x^+$, $a \in \Sigma$

(1) for $a \in \Sigma$ do $C_x^+ ← ∅$
(2) for $a \leftarrow σ$ downto 1 do
(3) for $i \in C_x^−$ in reverse order do // include elements added during the loop
(4) if $i > 0$ and $T[i] − 1 \leq a$ then pushback($i − 1, C_x^−(i−1)$)
(5) if $i > 0$ and $T[i] − 1 < a$ then $S_i \leftarrow S_i − 1$
(6) if $i \in C_x^+$ in reverse order do
(7) if $i > 0$ and $T[i] < a$ then pushfront($i − 1, C_x^−(i−1)$)

205

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

$F[i] = \min[k \in [i + 1..n)] | k \in C^+ \lor k = a$

$S_i = S_i + S_j$

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

$S_i = S_i + S_j$

Algorithm 4.28: For any $i, j \in [0..n)$, $T_i < T_j$ iff $S_i < S_j$ or $S_i = S_j$ and $T_i < T_j$.

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, $S_i > S_j$, but $S_i < S_j$ and thus $T_i < T_j$ by the claim. We will show that this is correct.

Let $i = F[j]$ and $k = i + 1 − j$. Then

- $T_i \in C−$ and thus $i − 1 \not= C−$. By Algorithm 4.25(b), $T[i − 1] > T[j]$.
- $T[i−1..k] = T[i−1..k]$ and thus $T[k−1] > T[i]$. 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 $a = T[i]$. Since $ε \in C^−$ and $k \in C^−$, we must have $T_i < a^− < T_j$.
- Since $T[i..k] = T[j..i]$ and $T_i < T_j$, we have $T_i < T_j$. 208
Algorithm 4.29: SAIS

Step 0: Choose C.
- Compute the types of suffixes. This can be done in $O(n)$ time based on Lemma 4.25.
- Set $C = \bigcup_{i \in C} C_i \cup \{n\}$. Note that $|C| \leq n/2$, since for all $i \in C$, $i - 1 \in C^C \subseteq C$.

Example 4.30:

$T = \text{mississippi}$

$C^*_i = \{2,5,8\}$, $C^*_m = C^*_p = C^*_s = \emptyset$, $C = \{2,5,8,14\}$

Step 2: Sort $T[0..n]$.
- Run InduceMinusSuffixes to construct the sorted lists $C^*_a$, $a \in [1..\sigma)$.
- Run InducePlusSuffixes to construct the lists $C^+_{\sigma-k}$.

The suffix array $SA = C^*_i C^+_i C^*_m C^*_p C^*_s = (14,13,12,8,9,5,2,1,0,11,10,7,4,6,3)$

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 $N_i \leq N_{i'}$ if $S'_i \leq S'_{i'}$.
- Construct the sequence $R = N_i N_{i'} \ldots N_0 \emptyset$, 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 4.31:

$R = [issi][issi][ippii][ippi][i] = 2210$, $SA_R = (3,2,1,0)$, $C^*_i = (8,5,2)$

Theorem 4.33: 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 $C^+_{\sigma-k}$, $a \in [1..\sigma)$.
3. Run InducePlusSuffixes to construct the lists $C^*_a$, $a \in [1..\sigma)$.
4. Remove non-$*$-type positions from $C^+_{\sigma-k} \ldots C^+_{\sigma-k-1}$.

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 R-divisort by Yuta Mori. It sorts the $*$-type suffixes non-recursively in $O(n \log n)$ 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.

Often augmented with additional data structures:

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

209

210

211

212

213