Overview. The paper is organized as follows. Section 3 explains the basic linear time algorithm DC3. We then use the concept of a difference cover introduced in Section 4 to describe a generalized algorithm called DC in Section 5 that leads to a space efficient algorithm in Section 6. Section 7 explains implementations of the DC3 algorithm in advanced models of computation. The results together with some open issues are discussed in Section 8.

2 Notation

We use the shorthands $[i, j] = \{i, \ldots, j\}$ and $[i, j) = [i, j-1]$ for ranges of integers and extend to substrings as seen below.

The **input** of a suffix array construction algorithm is a *string* $T = T[0, n) = t_0 t_1 \cdots t_{n-1}$ over the alphabet $[1, n]$, that is, a sequence of n integers from the range $[1, n]$. For convenience, we assume that $t_j = 0$ for $j \geq n$. Sometimes we also assume that $n + 1$ is a multiple of some constant v or a square to avoid a proliferation of trivial case distinctions and $\lceil \cdot \rceil$ operations. An implementation will either spell out the case distinctions or pad (sub)problems with an appropriate number of zero characters. The restriction to the alphabet $[1, n]$ is not a serious one. For a string T over any alphabet, we can first sort the characters of T, remove duplicates, assign a rank to each character, and construct a new string T' over the alphabet $[1, n]$ by renaming the characters of T with their ranks. Since the renaming is order preserving, the order of the suffixes does not change.

For $i \in [0, n]$, let S_i denote the *suffix* $T[i, n) = t_i t_{i+1} \cdots t_{n-1}$. We also extend the notation to sets: for $C \subseteq [0, n]$, $S_C = \{S_i \mid i \in C\}$. The goal is to sort the set $S_{[0,n]}$ of suffixes of T, where comparison of substrings or tuples assumes the lexicographic order throughout this paper. The **output** is the *suffix array* $SA[0, n]$ of T, a permutation of $[0, n]$ satisfying $S_{SA[0]} < S_{SA[1]} < \cdots < S_{SA[n]}$.

3 Linear-time algorithm

We begin with a detailed description of the simple linear-time algorithm, which we call DC3 (for Difference Cover modulo 3, see Section 4). A complete implementation in C++ is given in Appendix A. The execution of the algorithm is illustrated with the following example

$$T[0, n) = \begin{array}{cccccccccccc} \scriptstyle 0 & \scriptstyle 1 & \scriptstyle 2 & \scriptstyle 3 & \scriptstyle 4 & \scriptstyle 5 & \scriptstyle 6 & \scriptstyle 7 & \scriptstyle 8 & \scriptstyle 9 & \scriptstyle 10 & \scriptstyle 11 \\ \mathtt{y} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{d} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{d} & \mathtt{o} \end{array}$$

where we are looking for the suffix array

$$SA = (12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0) \ .$$

Step 0: Construct a sample. For $k = 0, 1, 2$, define

$$B_k = \{i \in [0, n] \mid i \bmod 3 = k\}.$$

Let $C = B_1 \cup B_2$ be the set of *sample positions* and S_C the set of *sample suffixes.*

Example. $B_1 = \{1, 4, 7, 10\}$, $B_2 = \{2, 5, 8, 11\}$, i.e., $C = \{1, 4, 7, 10, 2, 5, 8, 11\}$.

Step 1: Sort sample suffixes. For $k = 1, 2$, construct the strings

$$R_k = [t_k t_{k+1} t_{k+2}][t_{k+3} t_{k+4} t_{k+5}] \dots [t_{\max B_k} t_{\max B_k+1} t_{\max B_k+2}]$$

whose characters are triples $[t_i t_{i+1} t_{i+2}]$. Note that the last character of R_k is always unique because $t_{\max B_k+2} = 0$. Let $R = R_1 \odot R_2$ be the concatenation of R_1 and R_2. Then the (nonempty) suffixes of R correspond to the set S_C of sample suffixes: $[t_i t_{i+1} t_{i+2}][t_{i+3} t_{i+4} t_{i+5}] \dots$ corresponds to S_i. The correspondence is order preserving, i.e., by sorting the suffixes of R we get the order of the sample suffixes S_C.

Example. $R = [\mathtt{abb}][\mathtt{ada}][\mathtt{bba}][\mathtt{do0}][\mathtt{bba}][\mathtt{dab}][\mathtt{bad}][\mathtt{o00}]$.

To sort the suffixes of R, first radix sort the characters of R and rename them with their ranks to obtain the string R'. If all characters are different, the order of characters gives directly the order of suffixes. Otherwise, sort the suffixes of R' using Algorithm DC3.

Example. $R' = (1, 2, 4, 6, 4, 5, 3, 7)$ and $SA_{R'} = (8, 0, 1, 6, 4, 2, 5, 3, 7)$.

Once the sample suffixes are sorted, assign a rank to each suffix. For $i \in C$, let $rank(S_i)$ denote the rank of S_i in the sample set S_C. Additionally, define $rank(S_{n+1}) = rank(S_{n+2}) = 0$. For $i \in B_0$, $rank(S_i)$ is undefined.

Example.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$rank(S_i)$	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$	0	0

Step 2: Sort nonsample suffixes. Represent each nonsample suffix $S_i \in S_{B_0}$ with the pair $(t_i, rank(S_{i+1}))$. Note that $rank(S_{i+1})$ is always defined for $i \in B_0$. Clearly we have, for all $i, j \in B_0$,

$$S_i \le S_j \iff (t_i, rank(S_{i+1})) \le (t_j, rank(S_{j+1})).$$

The pairs $(t_i, rank(S_{i+1}))$ are then radix sorted.

Example. $S_{12} < S_6 < S_9 < S_3 < S_0$ because $(\mathtt{0}, 0) < (\mathtt{a}, 5) < (\mathtt{a}, 7) < (\mathtt{b}, 2) < (\mathtt{y}, 1)$.

Step 3: Merge. The two sorted sets of suffixes are merged using a standard comparison-based merging. To compare suffix $S_i \in S_C$ with $S_j \in S_{B_0}$, we distinguish two cases:

$$
\begin{aligned}
i \in B_1 : &\quad S_i \le S_j \iff (t_i, rank(S_{i+1})) \le (t_j, rank(S_{j+1})) \\
i \in B_2 : &\quad S_i \le S_j \iff (t_i, t_{i+1}, rank(S_{i+2})) \le (t_j, t_{j+1}, rank(S_{j+2}))
\end{aligned}
$$

Note that the ranks are defined in all cases.

Example. $S_1 < S_6$ because $(\texttt{a}, 4) < (\texttt{a}, 5)$ and $S_3 < S_8$ because $(\texttt{b}, \texttt{a}, 6) < (\texttt{b}, \texttt{a}, 7)$.

The time complexity is established by the following theorem.

Theorem 1. *The time complexity of Algorithm DC3 is $\mathcal{O}(n)$.*

Proof. Excluding the recursive call, everything can clearly be done in linear time. The recursion is on a string of length $\lceil 2n/3 \rceil$. Thus the time is given by the recurrence $T(n) = T(2n/3) + \mathcal{O}(n)$, whose solution is $T(n) = \mathcal{O}(n)$. □

4 Difference cover sample

The sample of suffixes in DC3 is a special case of a *difference cover sample*. In this section, we describe what difference cover samples are, and in the next section we give a general algorithm based on difference cover samples.

The sample used by the algorithms has to satisfy two *sample conditions*:

1. The sample itself can be sorted efficiently. Only certain special cases are known to satisfy this condition (see [37, 3, 9, 41] for examples). For example, a random sample would not work for this reason. Difference cover samples can be sorted efficiently because they are *periodic* (with a small period). Steps 0 and 1 of the general algorithm could be modified for sorting any periodic sample of size m with period length v in $\mathcal{O}(vm)$ time.

2. The sorted sample helps in sorting the set of all suffixes. The set of difference cover sample positions has the property that for any $i, j \in [0, n - v + 1]$ there is a small ℓ such that both $i + \ell$ and $j + \ell$ are sample positions. See Steps 2–4 in Section 5 for how this property is utilized in the algorithm.

The difference cover sample is based on difference covers [39, 10].

Definition 1. A set $D \subseteq [0, v)$ is a *difference cover* modulo v if

$$\{(i - j) \bmod v \mid i, j \in D\} = [0, v) \; .$$