8 Classical indexes

A *full-text index* for a string $T = t_1 t_2 \cdots t_n$ is a data structure that is built once, and that is kept in memory for answering an arbitrarily large number of queries on the position and frequency of *substrings* of T, with a time complexity that depends sublinearly on n. Consider for example a set $\mathcal{R} = \{R^1, R^2, \ldots, R^d\}$ of reads generated by a high-throughput sequencing machine from a genome G: assuming that each R^i is an exact copy of a substring of G, a routine question in read analysis is finding the position of each R^i in G. The lens of full-text indexes can also be used to detect common features of a set of strings, like their *longest common substring*: such analyses arise frequently in evolutionary studies, where substrings that occur exactly in a set of biological sequences might point to a common ancestral relationship.

The problem of matching and counting *exact substrings* might look artificial at a first glance: for example, reads in $\mathcal{R}$ are not exact substrings of G in practice. In the forthcoming chapters we will present more realistic queries and analyses, which will nonetheless build upon the combinatorial properties and construction algorithms detailed here. Although this chapter focuses on the fundamental data structures and design techniques of classical indexes, the presentation will be complemented with a number of applications to immediately convey the flexibility and power of these data structures.

8.1 k-mer index

The oldest and most popular index in *information retrieval* for natural-language texts is the so called *inverted file*. The idea is to sort in lexicographic order all the distinct words in a text T, and to precompute the set of occurrences of each word in T. A query on any word can then be answered by a binary search over the sorted list. Since biological sequences have no clear delimitation in words, we might use all the distinct substrings of T of a fixed length k. Such substrings are called *k-mers*.

DEFINITION 8.1 *The* k-mer index *(also known as the* q-gram index*) of a text* $T = t_1 t_2 \cdots t_n$ *is a data structure that represents* occurrence lists $\mathcal{L}(W)$ *for each* k*-mer* W *such that* $j \in \mathcal{L}(W)$ *iff* $T_{j..j+k-1} = W$.

For example, the k-mer index of string $T =$ AGAGCGAGAGCGCGC# for $k = 2$ is shown in Figure 8.1(a).

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

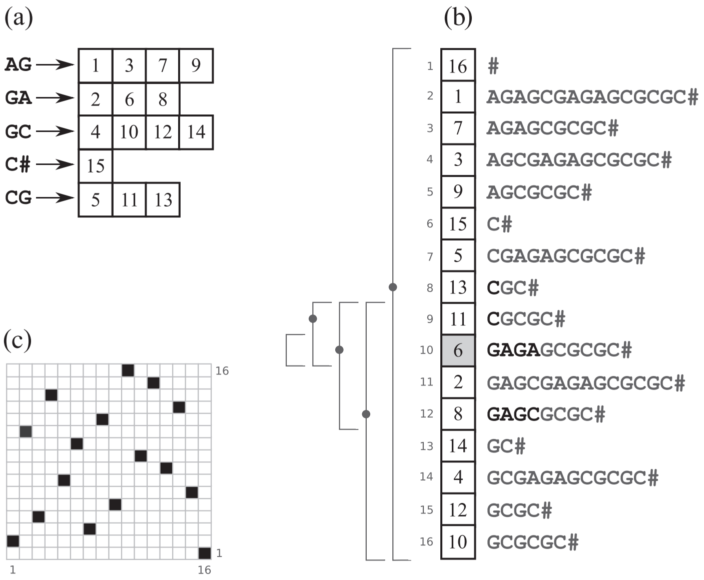

Figure 8.1 The 2-mer index (a) and the suffix array (b) of string $T =$ AGAGCGAGAGCGCGC#. Suffixes in (b) are shown just for clarity: they are not part of the suffix array. Intervals in panel (b) show the iterations of a binary search for the starting position of the interval of suffixes matching pattern $P =$ GAG. Note that P cannot be easily searched for in (a). The matrix in panel (c) visualizes the fact that SA_T is a permutation of $[1..|T|]$: cell (i,j) is black iff $\mathsf{SA}_T[i] = j$. Reading the matrix by row gives SA_T, and reading it by column gives the inverse suffix array of T.

Note that any substring P of length at most k can be searched for in the list of k-mers in $O(|P| \log n)$ time, and that its list of occurrences can be reconstructed by taking the union of the lists of all k-mers prefixed by P. However, it is not possible to do this efficiently for strings longer than k: we will waive this constraint in Section 8.2 using *suffix arrays*.

If each position is represented in $\log_2 n$ bits, the occurrence lists of all k-mers can take up exactly $n \log_2 n$ bits of space in total. Since the *difference* between consecutive numbers in the sorted occurrence lists is small, one could encode each difference using fewer than $\log_2 n$ bits. Concatenating such *variable-length* bit-fields into one long stream of bits makes decoding impossible, unless one encodes the lengths of the fields separately. Another option is to use *self-delimiting codes* that encode the length of each code internally. One well-known family of such codes is called *Elias codes*. Let $x^{\mathtt{bin}}$ denote the binary representation of an integer x. The Elias *gamma code* of $x > 0$ is defined as

$$\gamma(x) = \underbrace{\mathtt{0}\cdots\cdots\mathtt{0}}_{|x^{\mathtt{bin}}|-1 \text{ times}} \quad x^{\mathtt{bin}}$$

That is, the first run of zeros encodes the number of bits to represent $x^{\mathtt{bin}}$; with $x > 0$ the binary code $x^{\mathtt{bin}}$ starts always with 1 (if one needs to encode 0, one can use systematically $\gamma(x+1)$). For example, $\gamma(5) = \mathtt{00\,101}$, since $5^{\mathtt{bin}} = \mathtt{101}$. Note that a sequence of gamma-coded non-negative integers can be decoded in a unique way, and that $\gamma(x)$ takes exactly $2\lceil\log_2 x\rceil - 1$ bits. The idea of gamma codes can be applied recursively, to obtain the asymptotically smaller *delta codes* for $x > 0$:

$$\delta(x) = \underbrace{\mathtt{0}\cdots\cdots\cdots\cdots\mathtt{0}}_{|(|x^{\mathtt{bin}}|)^{\mathtt{bin}}|-1 \text{ times}}\ (|x^{\mathtt{bin}}|)^{\mathtt{bin}}\ x^{\mathtt{bin}}.$$

For example, $\delta(5) = \mathtt{0}^1\, 3^{\mathtt{bin}}\, 5^{\mathtt{bin}} = \mathtt{0\,11\,101}$.

Assume now that the sorted occurrence list $\mathcal{L}(W)$ for k-mer W starts with 0 (which is not encoded explicitly), and define $\mathtt{diff}_W[i] = \mathcal{L}(W)[i] - \mathcal{L}(W)[i-1]$ for $1 \le i \le |\mathcal{L}(W)|$. Then, the totality of gamma-coded lists takes the following number of bits:

$$\begin{aligned}
\sum_{W\in\Sigma^k}\sum_{i=1}^{|\mathcal{L}(W)|} |\gamma(\mathtt{diff}_W[i])| &= \sum_{W\in\Sigma^k}\sum_{i=1}^{|\mathcal{L}(W)|} (2\lceil\log(\mathtt{diff}_W[i])\rceil - 1)\\
&\le \sum_{W\in\Sigma^k}\sum_{i=1}^{|\mathcal{L}(W)|} (2\log(\mathtt{diff}_W[i]) + 1)\\
&\le n + 2\sum_{W\in\Sigma^k} |\mathcal{L}(W)|\log\left(\frac{n}{|\mathcal{L}(W)|}\right) \qquad (8.1)\\
&\le n + 2n\log\ell, \qquad (8.2)
\end{aligned}$$

where ℓ is the number of non-empty lists, that is, the number of distinct k-mers in T. Inequality (8.1) derives from the facts that $\sum_{i=1}^{|\mathcal{L}(W)|}\mathtt{diff}_W[i] \le n$ for each W and $\sum_{W\in\Sigma^k}|\mathcal{L}(W)| = n$, and application of Jensen's inequality

$$\frac{\sum_i a_i f(x_i)}{\sum_i a_i} \le f\left(\frac{\sum_i a_i x_i}{\sum_i a_i}\right), \qquad (8.3)$$

which holds for any concave function f. Here one should set $a_i = 1$ for all i, $x_i = \mathtt{diff}_W[i]$, and $f = \log$. Inequality (8.2) derives from the same facts, setting $a_i = |\mathcal{L}(W)|$, $x_i = n/|\mathcal{L}(W)|$, and $f = \log$ in Jensen's inequality.

THEOREM 8.2 *The k-mer index of a text $T = t_1t_2\cdots t_n \in \Sigma^*$ can be represented in $|\Sigma|^k(1+o(1)) + n(1+o(1)) + 2n\log\ell + \ell\log(n+2n\log\ell)$ bits of space such that each occurrence list $\mathcal{L}(W)$ can be decoded in $O(k+|\mathcal{L}(W)|)$ time, where ℓ is the number of non-empty occurrence lists.*

Proof See Exercise 8.1 on how to access the list corresponding to W in $O(k)$ time and Exercise 8.2 on how to encode the pointers to the non-empty elements using $|\Sigma|^k(1+o(1)) + \ell\log(n + 2n\log\ell)$ bits. It remains to show that each γ-coded value can be decoded in constant time. We use the same four Russians technique as in Chapter 3 to build a table of $\sqrt{n}$ entries storing for each bitvector of length $(\log n)/2$ the location of the first bit set. At most two accesses to this table are required for decoding a γ-coded value. □

See Exercises 8.3 and 8.4 for alternative ways to link k-mers to their occurrence lists.

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

8.2 Suffix array

The obvious problem with the k-mer index is the limitation to searching for patterns of fixed maximum length. This can be alleviated as follows. Consider all suffixes of string T that start with k-mer W. If one replaces W with those suffixes, and places the suffixes in lexicographic order, a binary search for the pattern can continue all the way to the end of each suffix if necessary. Listing all suffixes like this takes quadratic space, but it is also not required, since it suffices to store the starting positions of suffixes: see Figure 8.1(b).

DEFINITION 8.3 *The* suffix array $\mathsf{SA}_T[1..n]$ *of a text* $T = t_1t_2\cdots t_n$ *is the permutation of* $[1..n]$ *such that* $\mathsf{SA}_T[i] = j$ *iff suffix* $T_{j..n}$ *has position* i *in the list of all suffixes of* T *taken in lexicographic order.*

We say that suffix $T_{j..n}$ has *lexicographic rank* i iff $\mathsf{SA}_T[i] = j$. To ensure that suffixes can be sorted, we assume that $t_n = \#$ is an artificial character smaller than any other character in Σ, or in other words that $\mathsf{SA}_T[1] = n$. For example, the suffix array of T = AGAGCGAGAGCGCGC# is shown in Figure 8.1(b). We will frequently use the shorthand SA to mean SA_T when T is clear from the context, and we will call *inverse suffix array* an array $\mathsf{ISA}[1..n]$ such that $\mathsf{ISA}[\mathsf{SA}[j]] = j$ for all $j \in [1..n]$ (Figure 8.1(c)).

Let us now consider the binary search for the pattern in more detail. First observe that a substring P of T corresponds to a unique, contiguous *interval* $\overset{\bullet\!-\!\bullet}{P}$ in SA_T: the interval that contains precisely the suffixes of T that are prefixed by P. Knowing the interval $\overset{\bullet\!-\!\bullet}{P}$ allows one to derive in constant time the number of occurrences of P in T, and to output the occurrences themselves in time linear in the size of the output. Searching for P in SA_T corresponds thus to finding the starting and ending position of its interval, and this naturally maps to an $O(|P|\log n)$-time binary search over SA_T as illustrated in Example 8.1 and Figure 8.1(b).

Example 8.1 Let P = GAG and T = AGAGCGAGAGCGCGC#, and assume that we want to determine the starting position $\overset{\bullet\!-}{P}$ of $\overset{\bullet\!-\!\bullet}{P}$ in SA_T (see Figure 8.1(b)). We begin by considering the whole suffix array as a candidate region containing $\overset{\bullet\!-}{P}$; that is, we start from the interval $[1..16]$ in SA_T, and we compare $P\#_1$ with the suffix in the middle of the current search interval, that is, suffix $\mathsf{SA}[1 + \lfloor(16-1)/2\rfloor] = \mathsf{SA}[8] = 13$. Note that, since we are searching for the starting position of P, we append the artificial delimiter $\#_1$ to the end of P, where $\#_1$ is lexicographically smaller than any other character in $\Sigma \cup \{\#\}$. Symmetrically, searching for $\overset{-\!\bullet}{P}$ requires appending \$ to the end of P, where \$ is lexicographically larger than any other character in $\Sigma \cup \{\#_1, \#\}$. Since $P\#_1$ = GAG$\#_1$ is lexicographically larger than suffix $T_{13..16}$ = CGC#, we update the search interval to $[9..16]$ and we compare $P\#_1$ with suffix $\mathsf{SA}[9 + \lfloor(16-9)/2\rfloor] = \mathsf{SA}[12] = 8$ in the middle of the new search interval. $P\#_1$ = GAG$\#_1$ is lexicographically smaller than suffix $T_{8..16}$ = GAGCGCGC#, thus we update the search interval to $[9..12]$, and we compare $P\#_1$ with suffix $\mathsf{SA}[9 + \lfloor(12-9)/2\rfloor] = \mathsf{SA}[10] = 6$. Now $P\#_1$ = GAG$\#_1$ is lexicographically smaller than suffix $T_{6..16}$ = GAGAGCGCGC#, thus we update the search interval to

[9..10], and we compare $P\#_1$ with suffix $\mathsf{SA}[9+\lfloor(10-9)/2\rfloor] = \mathsf{SA}[9] = 11$. Since $P\#_1 = \mathtt{GAG\#_1}$ is lexicographically larger than suffix $T_{11..16} = \mathtt{CGCGC\#}$, we finally update the search interval to [10..10] and we stop, since the interval has reached size one. A symmetric binary search on $P\$ = \mathtt{GAG\$}$ results in $\overset{-\!\bullet}{P} = 12$. That is, $\overset{\bullet\!-\!\bullet}{\mathtt{GAG}} = [10..12]$.

THEOREM 8.4 *Given the suffix array $\mathsf{SA}_T[1..n]$ of a text $T = t_1t_2\cdots t_n$, and a pattern $P = p_1p_2\cdots p_m$, one can find in $O(m\log n)$ time the maximal interval $\overset{\bullet\!-\!\bullet}{P}$ in SA_T such that $T_{\mathsf{SA}[i]..\mathsf{SA}[i]+m-1} = P$ for all $i \in \overset{\bullet\!-\!\bullet}{P}$.*

The list of all suffixes of a string T in lexicographic order enjoys a remarkably regular structure. We will now detail some of its combinatorial properties and show how they can be leveraged for building SA_T from T.

8.2.1 Suffix and string sorting

It is not surprising that algorithms for building suffix arrays have deep connections to algorithms for sorting integers. We will first consider sorting fixed-length strings (all of length k) as an extension of sorting integers, then proceed to sorting variable-length strings, and finally show how these can be used as subroutines to sort the suffixes of a text.

Sorting a set of strings of fixed length

Let $L[1..n]$ be a sequence of integers $L[i] \in [1..\sigma]$ for $1 \leq i \leq n$. The folklore *counting sort* algorithm to sort L into L' such that $L'[i] \leq L'[i+1]$, for $1 \leq i \leq n-1$, works as follows. Build a vector $\mathtt{count}[1..\sigma]$ that stores in $\mathtt{count}[c]$ the number of entries i with value $L[i] = c$, with a linear scan over L. These counts can be turned into cumulative counts in another *block pointer* vector $B[1..\sigma]$ by setting $B[k] = B[k-1] + \mathtt{count}[k-1]$ for $k \in [2..\sigma]$, after initializing $B[1] = 1$. Finally, setting $L'[B[L[i]]] = L[i]$ and $B[L[i]] = B[L[i]] + 1$, for $i \in [1..n]$, produces the required output. Observe that before the iteration $B[c]$ points to the beginning of the block of characters c in the final L', and after the iteration $B[c]$ points to the beginning of the next block. For this analogy, let us call this step *block pointer iteration* of counting sort.

Assume now that L is a list of *pairs* with the ith pair denoted $(L[i].p, L[i].v)$, where p stands for *primary key* and v for *value*. One can verbatim modify the counting sort to produce L' such that $L'[i].p \leq L'[i+1].p$ for $1 \leq i \leq n-1$. For cases $L'[i].p = L[i+1].p$ one often wants to use the original entry as an *implicit secondary key*. This is accomplished automatically if i is processed left to right in the block pointer iteration of counting sort by setting $L'[B[L[i].p]] = L[i]$ and $B[L[i].p] = B[L[i].p] + 1$. Such sorted order is called *stable*.

Continuing the induction towards fixed-length strings, consider now that L is a list of *triplets* with the ith triplet denoted $(L[i].p, L[i].s, L[i].v)$, where we have added an *explicit secondary key* $L[i].s$. Obviously, we wish to stable sort L into L' first by the primary key and then by the secondary key, and this is easy to achieve by extending the counting sort into *radix sort*, as follows.

LEMMA 8.5 *Let $L[1..n]$ be a sequence of (primary key $L[i].p$, secondary key $L[i].s$, value $L[i].v$) triplets such that $L[i].p, L[i].s \in [1..\sigma]$ for $i \in [1..n]$. There is an algorithm to stable sort L into L' such that either $L'[i].p < L'[i+1].p$, or $L'[i].p = L'[i+1].p$ and $L'[i].s \leq L'[i+1].s$, for $i \in [1, n-1]$, in $O(\sigma + n)$ time and space.*

Proof Sort L independently into sequences L^p and L^s using stable counting sort with $L[i].p$ and $L[i].s$ as the sort keys, respectively. Let $B[1..\sigma]$ store the cumulative counts *before* the execution of the block pointer iteration of counting sort, when producing L^p. That is, $B[c]$ points to the first entry i of the final solution L' with primary key $L'[i].p = c$. The block pointer iteration is replaced by the setting of $L'[B[L^s[k].p]] = L^s[k]$ and $B[L^s[k].p] = B[L^s[k].p] + 1$ iterating k from 1 to n. Observe that here the explicit secondary keys replace the role of implicit secondary keys in the underlying stable counting sort. □

Consider now a sequence of strings $\mathcal{S} = W^1, \ldots, W^n \mid W^i \in [1..\sigma]^*$, and assume that we want to compute $\mathcal{S}^*$, the permutation of $\mathcal{S}$ in lexicographic order. Assuming the strings are all of some fixed length, the radix sort approach can be applied iteratively.

COROLLARY 8.6 *Let $\mathcal{S} = W^1, \ldots, W^n \mid W^i \in [1..\sigma]^*$ be a sequence of strings of fixed length k, that is, $k = |W^i|$ for all i. There is an algorithm to sort $\mathcal{S}$ into lexicographic order in $O(k(\sigma + n))$ time.*

Proof Apply Lemma 8.5 on $L = (w^1_1, w^1_2, 1), (w^2_1, w^2_2, 2), \ldots, (w^n_1, w^n_2, n)$ to produce L' such that $L'[j].v = i$ identifies the string W^i in $\mathcal{S}$ that has order j when considering only prefixes of length 2 in stable sorting. Let $R[1..n]$ store the *lexicographic ranks* of $L'[1..n]$ with $R[j]$ defined as the number of *distinct* primary and secondary element pairs occurring in prefix $L'[1..j]$. This array is computed by initializing $R[1] = 1$ and then setting $R[j] = R[j-1]$ if $L'[j].p = L'[j-1].p$ and $L'[j].s = L'[j-1].s$, otherwise $R[j] = R[j-1] + 1$, iterating j from 2 to n.

Now, form $L = (R[1], w_3^{L'[1].v}, L'[1].v), (R[2], w_3^{L'[2].v}, L'[2].v), \ldots, (R[n], w_3^{L'[n].v}, L'[n].v)$ and apply again Lemma 8.5 to produce L' such that $L'[j].v = i$ identifies the string W^i in $\mathcal{S}$ which has order j when considering only prefixes of length 3 in stable sorting. Then recompute lexicographic ranks $R[1..n]$ as above. Continuing this way until finally applying Lemma 8.5 on $L = (R[1], w_{k-1}^{L'[1].v}, L'[1].v), (R[2], w_k^{L'[2].v}, L'[2].v), \ldots, (R[n], w_k^{L'[n].v}, L'[n].v)$, one obtains L' such that $L'[j].v = i$ identifies the string W^i in $\mathcal{S}$ which has the order j when considering the full strings in stable sorting. The requested permutation of $\mathcal{S}$ in lexicographic order is obtained by taking $\mathcal{S}^*[j] = W^{L'[j].v}$. □

*Sorting a set of strings of variable length

Now let $\mathcal{S} = W^1, \ldots, W^n \mid W^i \in [1..\sigma]^*$ be a sequence of strings of *variable* length. We follow basically the same scheme as for strings of fixed length, except that we need to take into account the fact that, on moving from prefixes of length p to prefixes of length $p+1$, some strings can end. Filling tables of size $O(n)$ at each level p is not sufficient,

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

because the running time should be proportional to the sparse set of strings that are still alive at that level. We shall now proceed to the details on how to achieve the optimal running time by maintaining the sparse set of strings at each level.

Let m be $\max\{|W^k| : k \in [1..n]\}$, and for a given integer p consider the set of distinct prefixes $\mathcal{S}^p = \{W^k_{1..p} \mid k \in [1..n], p \in [1..m]\}$, where we assume that $W^k_{1..p} = W^k$ if $p > |W^k|$. We could sort $\mathcal{S}$ by induction over position $p \in [1..m]$: in particular, assuming that $\mathcal{S}$ is already sorted by $\mathcal{S}^{p-1}$ at iteration p, we could derive the lexicographic order induced by $\mathcal{S}^p$ from the lexicographic order of the characters that occur at position p (see Figure 8.2(a)). More formally, every string $W \in \mathcal{S}^p$ maps to a contiguous interval $\overleftrightarrow{W} = [\overleftarrow{W}..\overrightarrow{W}]$ in $\mathcal{S}^*$ that contains all strings in $\mathcal{S}$ that start with W, where $\overleftarrow{W}$ is the first position and $\overrightarrow{W}$ is the last position of the interval. Moreover, $\mathcal{I}^p = \{\overleftrightarrow{W} \mid W \in \mathcal{S}^p\}$ is a *partition* of $\mathcal{S}^*$, and partition $\mathcal{I}^p$ is a *refinement* of partition $\mathcal{I}^{p-1}$, in the sense that $\overleftrightarrow{Wa} \subseteq \overleftrightarrow{W}$ for any $a \in \Sigma$. We can thus transform $\mathcal{S}$ into $\mathcal{S}^*$ by iteratively assigning to each string W^k value $\overleftarrow{W^k_{1..p}}$ for increasing p, until all strings have distinct $\overleftarrow{W_{1..p}}$. The following lemma, illustrated in Figure 8.2, shows how to implement this idea using any algorithm for sorting triplets as a black box, as well as data structures analogous to those of counting sort.

LEMMA 8.7 *A sequence of strings $\mathcal{S} = W^1, \ldots, W^n \mid W^i \in [1..\sigma]^*$ can be sorted in lexicographic order in $O(\sigma + N)$ time, where $N = \sum_{k \in [1..n]} |W^k|$.*

Proof Consider the list L of all possible triplets $(\texttt{pos}, \texttt{char}, \texttt{str})$ for $\texttt{str} \in [1..n]$, $\texttt{pos} \in [1..m]$, and $\texttt{char} = W^{\texttt{str}}[\texttt{pos}]$. Sort L by its primary and secondary keys with Lemma 8.5. For clarity, denote by L^p the contiguous interval of the sorted L whose triplets have primary key p. Vectors L^p represent permutations of $\mathcal{S}$ sorted only by characters that occur at position p, and are thus homologous to the primary- and secondary-key vectors L^p and L^s used in Lemma 8.5. Note that each L^p is partitioned into contiguous intervals, each corresponding to triplets with the same character. In what follows, we will keep refining the sorted order from shorter prefixes to longer prefixes as in the case of fixed-length strings by iterating $p \in [1..m]$, but we will engineer each refinement to take $O(|L^p|)$ rather than $O(n)$ time.

Set $p = 1$, and assign to each triplet in L^1 the starting position of its corresponding interval (block) in L^1, that is, initialize a block pointer vector $B[1..n]$ so that $B[k] = |\{h : w^h_1 < w^k_1\}| + 1$ as in counting sort. In the generic iteration we are considering a specific value of $p > 1$, and we assume that $B[k]$ contains the starting position $\overleftarrow{W^k[1..p-1]}$ of the interval of string $W^k[1..p-1]$ in $\mathcal{S}^*$. Recall that every interval of $\mathcal{I}_p$ is completely contained inside an interval of $\mathcal{I}^{p-1}$, and that the existence of an interval $\overleftrightarrow{T} \in \mathcal{I}^{p-1}$ implies the existence of an interval $\overleftrightarrow{Ta} \in \mathcal{I}^p$ with $\overleftarrow{Ta} = \overleftarrow{T}$, where a is the smallest character at position p preceded by string T. Let $Q[1..n]$, $S[1..n]$, and $C[1..n]$ be arrays of temporary variables with the following meaning: Q records the current position inside an interval, $Q[\overleftarrow{T}] = \overleftarrow{Ta}$, S stores the current size of each interval, $S[\overleftarrow{T}] = |\overleftrightarrow{T}|$, and C stores the current character corresponding to each interval, $C[\overleftarrow{T}] = a$, for every interval $\overleftrightarrow{T} \in \mathcal{I}^{p-1}$ and for a *temporary* interval $\overleftrightarrow{Ta} \in \mathcal{I}^p$. Initialize Q in $O(|L^p|)$ time using two scans over L^p, as follows. In the first scan, consider the string $Ta = W^{L^p[k].\texttt{str}}[1..p]$

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

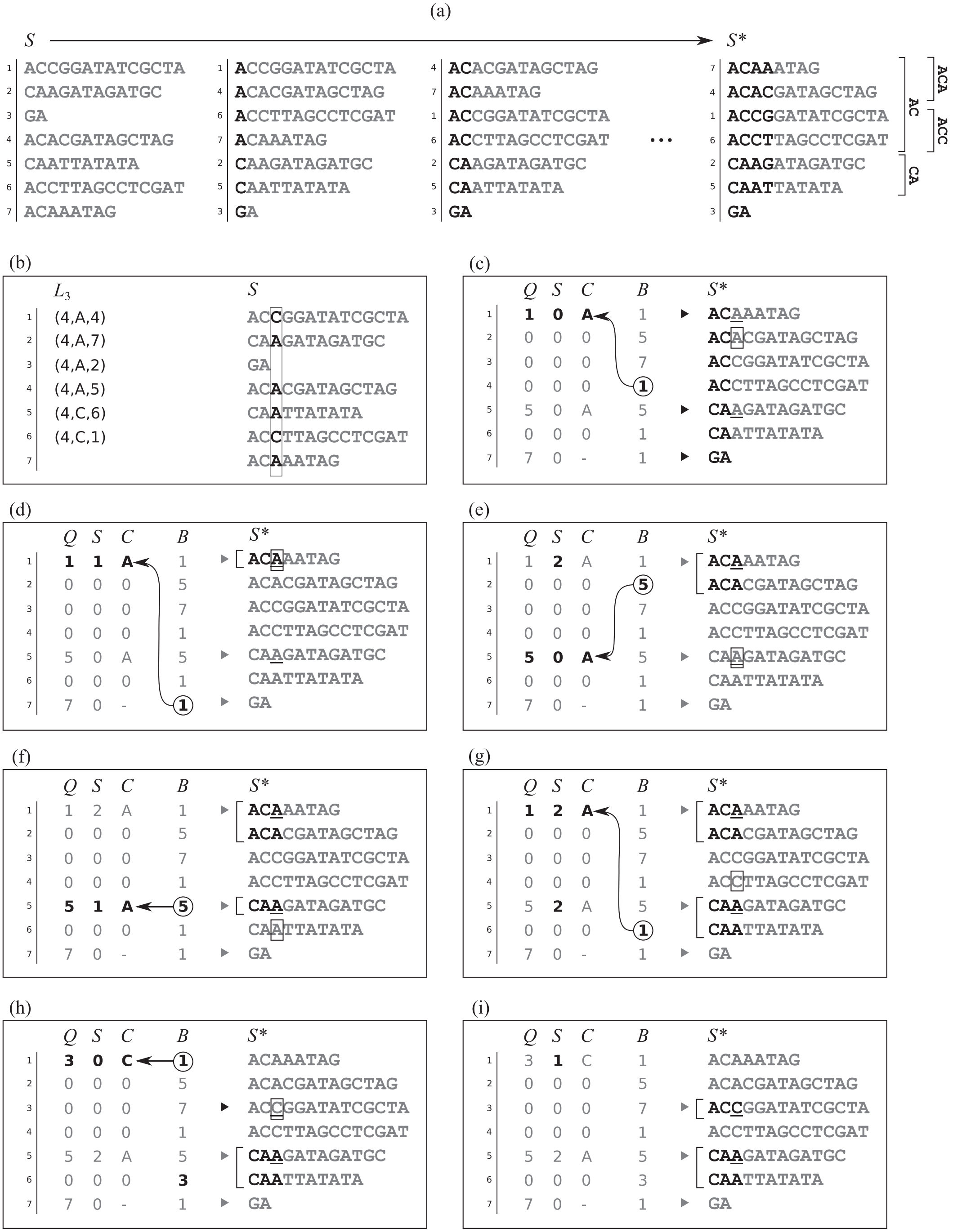

Figure 8.2 A sample run of the algorithm in Lemma 8.7. (a) The high-level strategy of the algorithm, with the intervals in $\mathcal{I}^2 \cup \mathcal{I}^3$ of size greater than one. (b) The initial sequence of strings $\mathcal{S}$ and the vector of triples L^3. (c–h) The data structures before processing $L^3[1], \ldots, L^3[6]$. (i) The data structures after processing $L^3[6]$. Triangles, square brackets, and underlined characters in $\mathcal{S}^*$ illustrate the meanings of Q, S, and C, respectively.

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

pointed by $L^p[k]$ for each $k \in [1..|L_p|]$, determine the starting position $\overset{\bullet\!-}{T}$ of the interval of T in $\mathcal{S}^*$ by setting $\overset{\bullet\!-}{T} = R[L^p[k].\mathtt{str}]$, and set $Q[\overset{\bullet\!-}{T}]$ to zero. In the second scan repeat the same steps for every $k \in [1..|L_p|]$, but if no Q value has yet been assigned to position $\overset{\bullet\!-}{T}$, that is, if $Q[\overset{\bullet\!-}{T}] = 0$, then set $Q[\overset{\bullet\!-}{T}]$ to $\overset{\bullet\!-}{T}$ itself, and initialize $C[\overset{\bullet\!-}{T}]$ to character $a = L_p[k].\mathtt{char}$. Note that a is the lexicographically smallest character that follows any occurrence of T as a prefix of a string of $\mathcal{S}$.

To compute in $O(|L^p|)$ time a new version of block pointer vector B that stores at $B[k]$ the starting position $W^k_{1..p}$ of the interval of string $W^k_{1..p}$ in $\mathcal{S}^*$, proceed as follows. Again, scan L^p sequentially, consider the string $Ta = W^{L^p[k].\mathtt{str}}_{1..p}$ pointed at by $L^p[k]$ for each $k \in [1..|L^p|]$, and compute $\overset{\bullet\!-}{T}$ by setting $\overset{\bullet\!-}{T} = B[L^p[k].\mathtt{str}]$. Recall that the values of $Q[\overset{\bullet\!-}{T}]$, $S[\overset{\bullet\!-}{T}]$, and $C[\overset{\bullet\!-}{T}]$ encode the starting position, size, and character of a sub-interval $\overset{\bullet\!-\!\bullet}{Tb}$ of $\overset{\bullet\!-\!\bullet}{T}$ for some $b \in \Sigma$. If character a equals b, that is, if $C[\overset{\bullet\!-}{T}] = L^p[k].\mathtt{char}$, string Ta belongs to the current sub-interval of $\overset{\bullet\!-\!\bullet}{T}$, so $B[L^p[k].\mathtt{str}]$ is assigned to $Q[\overset{\bullet\!-}{T}]$ and $S[\overset{\bullet\!-}{T}]$ is incremented by one. Otherwise, Ta does not belong to the current sub-interval of $\overset{\bullet\!-\!\bullet}{T}$, so we encode in $Q[\overset{\bullet\!-}{T}]$, $S[\overset{\bullet\!-}{T}]$ and $C[\overset{\bullet\!-}{T}]$ the new sub-interval $\overset{\bullet\!-\!\bullet}{Ta}$ by setting $Q[\overset{\bullet\!-}{T}] = Q[\overset{\bullet\!-}{T}]+S[\overset{\bullet\!-}{T}]$, $S[\overset{\bullet\!-}{T}] = 0$ and $C[\overset{\bullet\!-}{T}] = L^p[k].\mathtt{char}$. Then we set $B[L^p[k].\mathtt{str}] = Q[\overset{\bullet\!-}{T}]$. □

Suffix sorting by prefix doubling

Lemma 8.7 immediately suggests a way to build SA_T from T in $O(\sigma + |T|^2)$ time: consider the suffixes of T just as a set of strings, disregarding the fact that such strings are precisely the suffixes of T. But how does the set of suffixes of the same string differ from an arbitrary set of strings?

Denote again by $\mathcal{S}$ the set of suffixes of T, by $\mathcal{S}^*$ the permutation of $\mathcal{S}$ in lexicographic order, and by $\mathcal{S}_p$ the set of all distinct prefixes of length p of strings in $\mathcal{S}$. It is by now clear that every string $W \in \mathcal{S}_2$ maps to a contiguous interval $\overset{\bullet\!-\!\bullet}{W}$ of $\mathcal{S}^*$, and thus of SA_T. Assume again that we have an approximation of SA_T, in which such intervals are in lexicographic order, but suffixes inside the same interval are not necessarily in lexicographic order. Denote this permutation by SA^2. Clearly, SA^2 contains enough information to build a refined permutation SA^4 in which suffixes in the same interval of SA^2 are sorted according to their prefix of length four. Indeed, if $T_{i..n}$ and $T_{j..n}$ are such that $T_{i..i+1} = T_{j..j+1}$, then the lexicographic order of substrings $T_{i..i+3}$ and $T_{j..j+3}$ is determined by the order of $T_{i+2..i+3}$ and $T_{j+2..j+3}$ in SA^2. The idea of the *prefix-doubling technique* for suffix array construction consists in iterating this inference $O(\log|T|)$ times, progressively refining SA^2 into SA by considering prefixes of exponentially increasing length of all suffixes of T.

LEMMA 8.8 *There is a $O((\sigma + n)\log n)$-time algorithm to construct SA_T from T.*

Proof Sort the list of triples $L = (t_1, t_2, 1), (t_2, t_3, 2), \ldots, (t_n, t_{n+1}, n)$ into a new list L' in $O(\sigma + n)$ time, and assign to each suffix $T_{k..n+1}$ the starting position $\overset{\bullet\!-}{T_{k..k+1}}$ of the interval of its length-2 prefix $T_{k..k+1}$ in SA^2, as follows. Initialize an array $R[1..2n]$ with ones. Scan L' and set $R[L'[k].v] = R[L'[k-1].v]$ if $L'[k].p = L'[k-1].p$ and $L'[k].s = L'[k-1].s$, otherwise set $R[L'[k].v] = R[L'[k-1].v] + 1$. Values in $R[n+1..2n]$ are left as one, and will not be altered by the algorithm. At this point, refining SA^2 into SA^4

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

requires just sorting a new list $L = (R[1], R[3], 1), (R[2], R[4], 2), \ldots, (R[n], R[n+2], n)$, that is, sorting suffixes using as primary key their prefix of length two and as secondary key the substring of length two that immediately follows their length-two prefix. Having all values in $R[n+1..2n]$ constantly set to one guarantees correctness. Updating R and repeating this process $O(\log n)$ times produces SA. Indeed, the pth step of the algorithm sorts a list $L = (R[1], R[1+2^p], 1), (R[2], R[2+2^p], 2), \ldots, (R[n], R[n+2^p], n)$, where $R[k]$ contains the starting position $\overline{T[k..k+2^p]}$ of the interval of string $T[k..k+2^p]$ in SA. Thus, after the pth step, $R[k]$ is updated to the starting position $\overline{T[k..k+2^{p+1}]}$ of the interval of string $T[k..k+2^{p+1}]$ in SA. Clearly, when $2^p \geq n$ every cell of R contains a distinct value, the algorithm can stop, and $\mathsf{SA}[R[k]] = k$. □

Suffix sorting in linear time

An alternative strategy for building SA_T could be to iteratively fill a sparse version $\widetilde{\mathsf{SA}}_T$ initialized with just some seed values (see Figure 8.3). More specifically, let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\} \subseteq [1..n]$ be a set of positions in T, and assume that $\widetilde{\mathsf{SA}}[k] = \mathsf{SA}[k]$ for $\mathsf{SA}[k] \in \mathcal{I}$, and that $\widetilde{\mathsf{SA}}[k] = 0$ otherwise. Again, store in vector $Q[c]$ the position of the smallest zero in $\widetilde{\mathsf{SA}}[\overleftarrow{c}..\overrightarrow{c}]$ for every $c \in [1..\sigma]$. Assume that we are at position k in $\widetilde{\mathsf{SA}}$, and that we have computed the value of $\widetilde{\mathsf{SA}}[h]$ for all $h \in [1..k]$. Consider the *previous suffix* $T[\widetilde{\mathsf{SA}}[k]-1..n]$ and set $c = T[\widetilde{\mathsf{SA}}[k]]$. If $T[\widetilde{\mathsf{SA}}[k]-1..n]$ is lexicographically

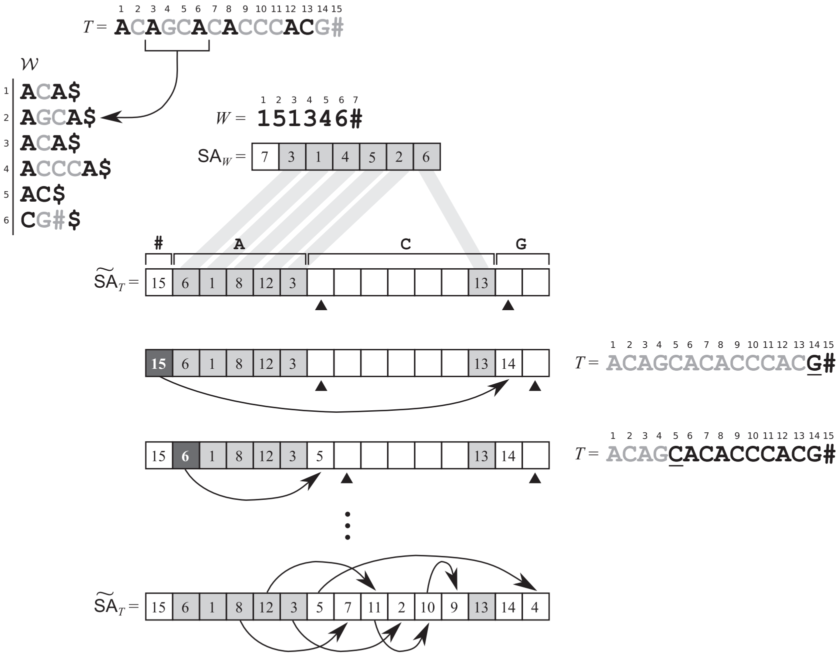

Figure 8.3 A strategy for the linear-time construction of SA_T. Set $\mathcal{I}$ of small suffixes is marked by black letters in the top-most occurrence of T, and by gray cells in $\widetilde{\mathsf{SA}}_T$. Triangles signal the current content of array Q. Note that substring ACA\$ occurs twice in $\mathcal{W}$, and that substring AC\$ occurs once.

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

smaller than the current suffix $T[\widetilde{\mathsf{SA}}[k]..n]$, then it already appears at an index smaller than k in $\widetilde{\mathsf{SA}}$. Otherwise, we can safely append suffix $T[\widetilde{\mathsf{SA}}[k]-1..n]$ to the current list of suffixes that start with c, that is, we can set $\widetilde{\mathsf{SA}}[Q[c]]$ to $\widetilde{\mathsf{SA}}[k]-1$ and we can increment pointer $Q[c]$ by one. Clearly this $O(n)$ scan of $\widetilde{\mathsf{SA}}$ works only if $\widetilde{\mathsf{SA}}[k] \neq 0$ at the current position k, and this holds either because $\mathsf{SA}[k] \in \mathcal{I}$ or because suffix $T[\mathsf{SA}[k]+1..n]$ has already been processed by the algorithm; that is, because the current suffix $T[\mathsf{SA}[k]..n]$ is *lexicographically larger* than suffix $T[\mathsf{SA}[k]+1..n]$. This motivates the definition of *small* and *large suffixes*.

DEFINITION 8.9 *A suffix $T_{i..n}$ of a string T is* large *(respectively,* small*) if $T_{i..n}$ is lexicographically larger (respectively, smaller) than $T_{i+1..n}$.*

It follows that, if $\mathcal{I}$ is the set of starting positions of all *small* suffixes of T, and if $\widetilde{\mathsf{SA}}$ is initialized according to $\mathcal{I}$, we can fill all the missing values of $\widetilde{\mathsf{SA}}$ in a single scan, obtaining SA: see Exercise 8.12.

Small and large suffixes enjoy a number of combinatorial properties: we focus on small suffixes here, with the proviso that everything we describe for small suffixes can easily be mirrored for large suffixes. Note that, in order to initialize $\widetilde{\mathsf{SA}}$, we need at least a Boolean vector $I[1..n]$ such that $I[k] = 1$ iff suffix $T_{i..n}$ is small. This vector can easily be computed in $O(n)$ time: see Exercise 8.10. Assume thus that $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ is known through I, that $\mathcal{S}$ is the sequence of suffixes $T[i_1..n], T[i_2..n], \ldots, T[i_m..n]$, and that $\mathcal{S}^*$ is the permutation of $\mathcal{S}$ in lexicographic order. In particular, let vector R store in $R[k]$ the position of suffix $T[i_k..n]$ in $\mathcal{S}^*$ and assume $R[m+1] = \#$. Note that we can initialize the entries of $\widetilde{\mathsf{SA}}$ that correspond to all small suffixes of T by scanning the suffix array SA_R *of array R considered as a string*, in overall $O(n)$ time: see Exercise 8.11. The bottleneck to building SA_T becomes thus computing SA_R.

Consider again set $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$: intuitively, most of the information in R is already determined by the substrings $T[i_1..i_2], T[i_2..i_3], \ldots, T[i_m..n]$. We might thus try to compute SA_R by sorting such substrings and by extrapolating SA_R from the sorted list. Specifically, consider the sequence of substrings $\mathcal{W} = T[i_1..i_2]\$, T[i_2..i_3]\$, \ldots, T[i_m..n]\$$ of total length $O(n)$, where the artificial separator \$ is lexicographically larger than σ, and denote by $\mathcal{W}^*$ the permutation of $\mathcal{W}$ in lexicographic order. Recall that we can build $\mathcal{W}^*$ from $\mathcal{W}$ in $O(\sigma + n)$ time using Lemma 8.7. In particular, the lemma returns a vector W such that $W[k]$ is the position of string $T[i_k..i_{k+1}]\$$ in $\mathcal{W}^*$ (see Figure 8.3), where again we assume $W[m+1] = \#$. Our plan is to infer SA_R from SA_W by exploiting another property of small suffixes.

LEMMA 8.10 *Let $\overset{\bullet\!-\!\bullet}{c}$ be the interval of the single-character string $c \in [1..\sigma]$ in SA_T. In $\overset{\bullet\!-\!\bullet}{c}$, large suffixes occur before small suffixes, for any $c \in [1..\sigma]$.*

Proof Let $T_{i..n}$ and $T_{j..n}$ be two suffixes of T such that $t_i = t_j = c$, and assume that $T_{i..n}$ is a small suffix and $T_{j..n}$ is a large suffix. Then $T_{i..n} = c^p dv\#$ with $p \geq 1$, $d \in [c+1..\sigma]$, and $v \in \Sigma^*$, and $T_{j..n} = c^q bw\#$ with $q \geq 1$, $b < c$, and $w \in \Sigma^*$. Thus, $T_{j..n}$ is lexicographically smaller than $T_{i..n}$. □

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

Note in particular that interval $\overset{\bullet\!-\!\bullet}{c}$ with $c = 1$ contains only small suffixes, and that interval $\overset{\bullet\!-\!\bullet}{c}$ with $c = \sigma$ contains only large suffixes. As anticipated, Lemma 8.10 is the bridge between SA_W and SA_R:

LEMMA 8.11 $\mathsf{SA}_R = \mathsf{SA}_W$.

Proof Consider two suffixes $T[i_k..n]$ and $T[i_h..n]$ with i_k and i_h in $\mathcal{I}$: we prove that, if i_k follows i_h in SA_W, then i_k follows i_h also in SA_R. Assume that i_k follows i_h in SA_W and substring $S\$ = T[i_k..i_{k+1}]\$$ is different from substring $S'\$ = T[i_h..i_{h+1}]\$$. This could happen because a prefix $S_{1..p}$ with $p \leq |S|$ is not a prefix of S': in this case, it is clear that $T[i_k..n]$ follows $T[i_h..n]$ in SA_R as well. Alternatively, it might be that S is a prefix of S', but $S\$$ is not (for example, for strings $S = \mathtt{AC}$ and $S' = \mathtt{ACA}$ in Figure 8.3). In this case, character $S'_{|S|}$ equals the last character of S: let c be this character, and let $S = Vc$ for $V \in \Sigma^*$. Then, suffix $T[i_k..n]$ equals $V \cdot T[i_{k+1}..n]$ and suffix $T[i_h..n]$ equals $V \cdot T[i_{h+|S|-1}..n]$. Note that both suffix $T[i_{k+1}..n]$ and suffix $T[i_{h+|S|-1}..n]$ start with character c, but $i_{h+|S|-1}$ does not belong to $\mathcal{I}$, thus, by Lemma 8.10, suffix $T[i_{h+|S|-1}..n]$ is lexicographically smaller than suffix $T[i_{k+1}..n]$. It follows that suffix $T[i_h..n]$ is lexicographically smaller than suffix $T[i_k..n]$ as well.

Assume now that i_k follows i_h in SA_W, but $S\$ = T[i_k..i_{k+1}]\$$ equals $S'\$ = T[i_h..i_{h+1}]\$$, i.e. $W[k] = W[h]$ (for example, for $S = S' = \mathtt{ACA}$ in Figure 8.3). Thus, the sequence of numbers $W[k+1..m]\#$ follows the sequence of numbers $W[h+1..m]\#$ in lexicographic order, i.e. $T[i_{k+x}..i_{k+x+1}]$ follows $T[i_{h+x}..i_{h+x+1}]$ in lexicographic order for some $x \geq 1$, and $T[i_{k+y}..i_{k+y+1}]$ equals $T[i_{h+y}..i_{h+y+1}]$ for all $y < x$. This implies that suffix $T[i_k..n]$ follows suffix $T[i_h..n]$ in lexicographic order. □

It thus remains to compute SA_W. If $|\mathcal{I}| \leq n/2$, i.e. if T has at most $n/2$ small suffixes, we could apply the same sequence of steps recursively *to* W, i.e. we could detect the small suffixes of W, split W into substrings, sort such substrings, infer the lexicographic order of all small suffixes of W, and then extrapolate the lexicographic order of all suffixes of W. If T has more than $n/2$ small suffixes, we could recurse on large suffixes instead, with symmetrical arguments. This process takes time:

$$\begin{aligned} f(n,\sigma) &= an + b\sigma + f(n/2, n/2) \\ &= an + b\sigma + an/2 + bn/2 + an/4 + bn/4 + \cdots + a + b \\ &\in O(\sigma + n), \end{aligned}$$

where a and b are constants. We are thus able to prove the main theorem of this section, as follows.

THEOREM 8.12 *The suffix array* SA_T *of a string* T *of length* n *can be built in* $O(\sigma + n)$ *time.*

8.3 Suffix tree

A *trie* is a labeled tree $\mathcal{T}$ that represents a set of strings. Specifically, given a set of variable-length strings $\mathcal{S} = \{S^1, \ldots, S^n\}$ over alphabet $\Sigma = [1..\sigma]$, the *trie of* $\mathcal{S}$ is a tree

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

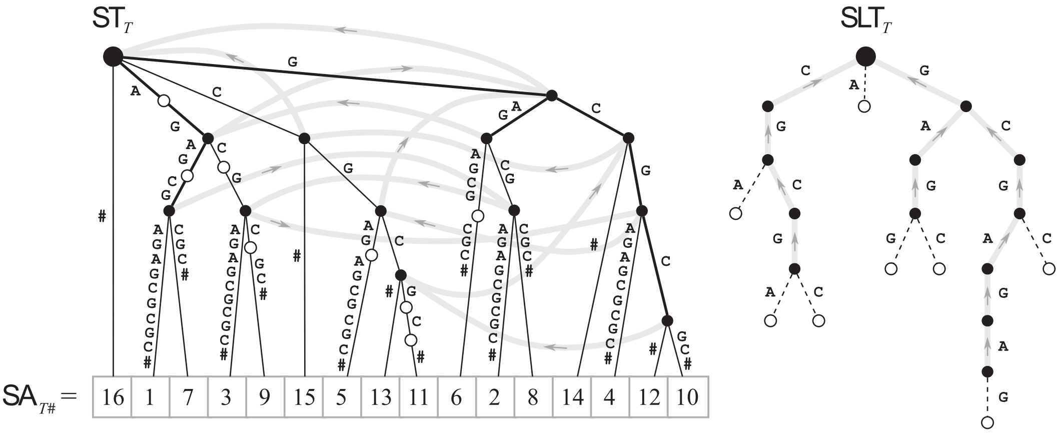

Figure 8.4 Relationships among the suffix tree, the suffix-link tree, and the suffix array of string $T =$ AGAGCGAGAGCGCGC. Thin black lines: edges of ST$_T$; thick gray lines: suffix links; thin dashed lines: implicit Weiner links; thick black lines: the subtree of ST$_T$ induced by maximal repeats. Black dots: nodes of ST$_T$; large black dot: r; white dots: destinations of implicit Weiner links; squares: leaves of ST$_T$; numbers: starting position of each suffix in T. For clarity, implicit Weiner links are not overlaid to ST$_T$, and suffix links from the leaves of ST$_T$ are not drawn.

$\mathcal{T}$ with at most n leaves and exactly n marked nodes, such that (1) every edge (v_i, v_j) is labeled by a character $\ell(v_i, v_j) \in \Sigma$; (2) the edges that connect a node to its children have distinct labels; (3) every leaf is marked; and (4) every path from the root to a marked node spells a distinct string of $\mathcal{S}$. A *compact trie*, or *Patricia trie*, is a tree $\mathcal{T}'$ whose edges are labeled by *strings* in Σ^+, rather than by single characters, and it can be thought of as obtained from a trie $\mathcal{T}$ as follows: for every path $v_1, v_2, \ldots, v_k$ such that v_i has only one child for every $i \in [1..k-1]$, remove nodes $v_2, v_3, \ldots, v_{k-1}$ and the incoming edges to $v_2, v_3, \ldots, v_k$, and add edge (v_1, v_k) with label $\ell(v_1, v_2) \cdot \ell(v_2, v_3) \cdot \cdots \cdot \ell(v_{k-1}, v_k)$. Clearly $\mathcal{T}'$ has at most $n-1$ internal nodes, where n is the number of leaves in $\mathcal{T}$.

The *suffix tree* is one of the oldest and most powerful full-text indexes, and it can be thought of as a compact trie built on the set of all *suffixes* of a string, or equivalently as a search tree built on top of the suffix array, as visualized in Figure 8.4.

DEFINITION 8.13 *The suffix tree* $\mathsf{ST}_T = (V, E)$ *of a string* $T \in [1..\sigma]^n$ *is a tree rooted in* $r \in V$. *The edges are labeled by substrings of T#. The leaves of the tree correspond to suffixes of T#, and the path from the root to a leaf spells the corresponding suffix. The label of each edge is of maximal length, so that the tree branches at each internal node, and the labels that connect a node to its children start with distinct characters.*

We assume that the children of a node v are ordered lexicographically according to the labels of the corresponding edges. We denote by $\ell(e)$ the label of an edge $e \in E$ and by $\ell(v)$ the string $\ell(r, v_1) \cdot \ell(v_1, v_2) \cdot \cdots \cdot \ell(v_{k-1}, v)$, where $r, v_1, v_2, \ldots, v_{k-1}, v$ is the path of v in ST$_T$. We say that node v has *string depth* $|\ell(v)|$. Since the tree has exactly $n+1$ leaves, and each internal node is branching, there are at most n internal nodes. Observe that the edge labels can be stored not as strings, but as pointers to the corresponding substring of T#: see Figure 8.4. Hence, each node and edge will store a

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

constant number of values or pointers, and each such value or pointer can be encoded using $\log n$ bits. A careful implementation requires $(3|V|+2n)\log n$ bits, without pointers from child to parent and other auxiliary information.

Searching T for a pattern $P = p_1p_2 \dots p_k$ clearly amounts to following the path labeled by P from r in ST_T and performing a binary search at each internal node. Indeed, if P is a substring of T, then it is the prefix of some suffix of T; thus it is a prefix of a label $\ell(v)$ for some node $v \in V$. For example, in Figure 8.4 $P = \mathtt{AGC}$ is a prefix of suffixes starting $\mathtt{AGCG}$. We call node v the *proper locus* of P if the search for P ends at an edge (w, v). In our example, the search for $P = \mathtt{AGC}$ ends at an edge (w, v) such that $\ell(w) = \mathtt{AG}$ and $\ell(v) = \mathtt{AGCG}$. Then, the leaves in the subtree rooted at v are exactly the starting positions of P in T. In our example, leaves under v point to the suffixes $T_{3..n}$ and $T_{9..n}$, which are the only suffixes prefixed by $P = \mathtt{AGC}$. This search takes $O(|P|\log\sigma + |\mathcal{L}_T(P)|)$ time, where $\mathcal{L}_T(P)$ is the set of all starting positions of P in T.

8.3.1 Properties of the suffix tree

In the forthcoming chapters we shall be heavily exploiting some key features of suffix trees. This is a good point to introduce those notions, although we are not using all of these features yet; the reader might opt to come back to some of these notions as required when they arise.

Building and using ST_T often benefits from an additional set of edges, called *suffix links*. Let the label $\ell(v)$ of the path from the root to a node $v \in V$ be aX, with $a \in \Sigma$. Since string X occurs at all places where aX occurs, there must be a node $w \in V$ with $\ell(w) = X$; otherwise v would not be a node in V. We say that there is a *suffix link from v to w labeled by a*, and we write $sl(v) = w$. More generally, we say that the set of labels $\{\ell(v) \mid v \in V\}$ enjoys the *suffix closure* property, in the sense that, if a string W belongs to this set, so does every one of its suffixes. Note that, if v is a leaf, then $sl(v)$ is either a leaf or r. We denote the set of all suffix links of ST_T by $L = \{(v, sl(v), a) \mid v \in V, sl(v) \in V, \ell(v) = a\ell(sl(v)), a \in \Sigma\}$. It is easy to see that the pair (V, L) is a trie rooted at r: we call such a trie the *suffix-link tree* SLT_T of string T: see Figure 8.4.

Inverting the direction of all suffix links yields the so-called *explicit Weiner links*, denoted by $\underline{L} = \{(sl(v), v, a) \mid v \in V, sl(v) \in V, \ell(v) = a\ell(sl(v)), a \in \Sigma\}$. Clearly a node in V might have more than one outgoing Weiner link, and all such Weiner links have different labels. Given a node v and a character $a \in \Sigma$, it might happen that string $a\ell(v)$ does occur in T, but that it does not label any node in V: we call all such extensions of nodes in V *implicit Weiner links*, and we denote the set of all implicit Weiner links by $\underline{L}'$: see Figure 8.4. Note that Weiner links from leaves are explicit.

We summarize the following characteristics of suffix trees.

OBSERVATION 8.14 *The numbers of suffix links, explicit Weiner links, and implicit Weiner links in the suffix tree ST_T of string $T = t_1t_2 \cdots t_{n-1}$ are upper bounded by $2n - 2$, $2n - 2$, and $4n - 3$, respectively.*

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

Proof Each of the at most $2n-2$ nodes of a suffix tree (other than the root) has a suffix link. Each explicit Weiner link is an inverse of a suffix link, so their total number is also at most $2n-2$.

Consider a node v with only one implicit Weiner link $e = (\ell(v), a\ell(v))$ in ST_T. The number of such nodes, and thus the number of such implicit Weiner links, is bounded by $2n-1$. Call these the implicit Weiner links of class I, and the remaining ones the implicit Weiner links of class II. Consider then a node v with two or more (class II) implicit Weiner links forming set $W_v = \{c \mid e_c = (\ell(v), c\ell(v))$ is a Weiner link in $\mathsf{ST}_T\}$. This implies the existence of a node w in the suffix tree of the *reverse* of T, $\mathsf{ST}_{\underline{T}}$, labeled by the reverse of $\ell(v)$, denoted by $\underline{\ell(v)}$; each $c \in W_v$ can be mapped to a distinct edge of $\mathsf{ST}_{\underline{T}}$ connecting w to one of its children. This is an injective mapping from class II implicit Weiner links to the at most $2n-2$ edges of the suffix tree of $\underline{T}$. The sum of class I and II, that is, all implicit Weiner links, is hence bounded by $4n-3$. □

The bound on the number of implicit Weiner links can easily be improved to $2n-2$: see Exercise 8.18. With more involved arguments, the bound can be further improved to n: see Exercise 8.19.

8.3.2 Construction of the suffix tree

Despite its invention being anterior to the suffix array, the suffix tree can be seen as an extension of it: building ST_T from SA_T clarifies the deep connections between these two data structures. Throughout this section we assume that $T = t_1t_2\cdots t_n$ with $t_n = \#$, and we assume that we have access both to $\mathsf{SA}_T[1..n]$ and to an additional *longest common prefix array* $\mathsf{LCP}[2..n]$ such that $\mathsf{LCP}[i]$ stores the length of the longest prefix that is common to suffixes $T[\mathsf{SA}[i-1]..n]$ and $T[\mathsf{SA}[i]..n]$. Building LCP_T in $O(n)$ time from SA_T is left to Exercise 8.14. To build ST, we first insert the suffixes of T as leaves in lexicographic order, i.e. we assign suffix $T[\mathsf{SA}[i]..n]$ to leaf i. Then, we build the internal nodes by iterating over LCP_T from left to right, as follows.

Consider a temporary approximation $\widetilde{\mathsf{ST}}^{i-1}$ of ST, which is built using only suffixes $T[\mathsf{SA}[1]..n]$, $T[\mathsf{SA}[2]..n]$, $\ldots$, $T[\mathsf{SA}[i-1]..n]$. In order to update this tree to contain also suffix $T[\mathsf{SA}[i]..n]$, we can traverse $\widetilde{\mathsf{ST}}^{i-1}$ bottom-up starting from its right-most leaf, until we find a node v such that $|\ell(v)| \leq \mathsf{LCP}[i]$. If $|\ell(v)| = \mathsf{LCP}[i]$ we insert a new edge from v to leaf i, otherwise we split the right-most edge from v, say (v, w), at depth $\mathsf{LCP}[i] - |\ell(v)|$, and create a new internal node with one child pointing to node w and another child pointing to leaf i (Figures 8.5(b) and (c)). Clearly, after scanning the whole LCP we have $\widetilde{\mathsf{ST}}^{n} = \mathsf{ST}$. Observe that, if we traverse from v to its parent during the traversal, we never visit v again. Each traversal to add a new suffix ends with a constant number of operations to modify the tree. Thus the total traversal and update time can be amortized to the size of ST, and we obtain the following result.

THEOREM 8.15 *The suffix tree* ST_T *of a string* $T = t_1t_2\cdots t_{n-1} \in [1..\sigma]^{n-1}\#$ *can be built in time* $O(\sigma + n)$ *so that the leaves of* ST_T *are sorted in lexicographic order.*

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

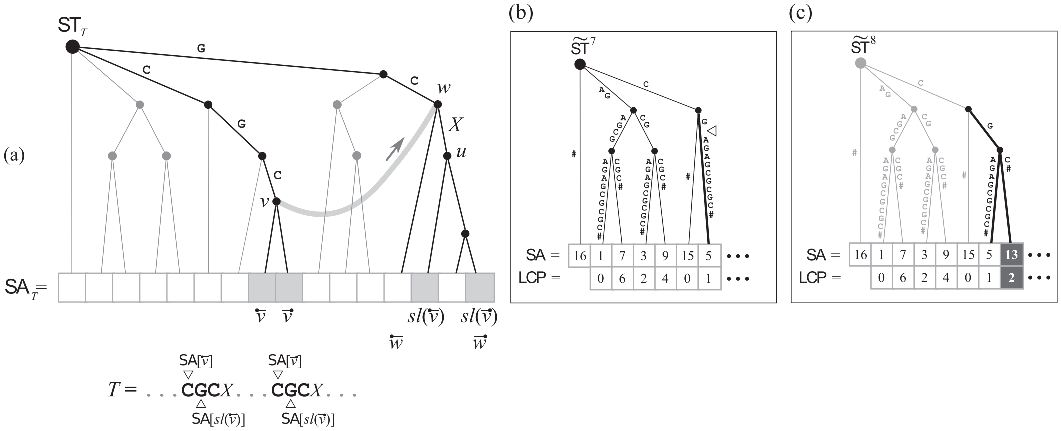
(a)
ST_T
G
C
C
G
C
w
X
u
v
$\mathsf{SA}_T=$
$\overleftarrow{v}$ $\overrightarrow{v}$
$sl(\overleftarrow{v})$ $sl(\overrightarrow{v})$
$\overleftarrow{w}$ $\overrightarrow{w}$
$\mathsf{SA}[\overleftarrow{v}]$ $\mathsf{SA}[\overrightarrow{v}]$
$T = \ldots \mathsf{CGC}X \ldots \mathsf{CGC}X \ldots$
$\mathsf{SA}[sl(\overleftarrow{v})]$ $\mathsf{SA}[sl(\overrightarrow{v})]$
(b)
$\widetilde{\mathsf{ST}}^7$
SA = 16 1 7 3 9 15 5 •••
LCP = 0 6 2 4 0 1 •••
(c)
$\widetilde{\mathsf{ST}}^8$
SA = 16 1 7 3 9 15 5 13 •••
LCP = 0 6 2 4 0 1 2 •••

Figure 8.5 (a) Visualizing Lemma 8.16. (b,c) Building the suffix tree from the suffix array and the LCP array: the current approximation $\widetilde{\mathsf{ST}}^7$ (b) and the new approximation $\widetilde{\mathsf{ST}}^8$ (c).

Adding suffix links to the leaves of ST_T using the suffix array and the inverse suffix array of T is easy: see Exercise 8.16. But how can we augment ST_T with the suffix links of internal nodes?

Let $Y \subset V$ be the set of leaves of ST_T. Note that every internal node of ST_T corresponds to an interval $\overleftrightarrow{v}$ in SA_T, where $\overleftarrow{v}$ and $\overrightarrow{v}$ are the leaves with lexicographically smallest and largest labels in the subtree of ST_T rooted at v. Note also that, if node v is an ancestor of node w in ST_T, then $\overleftrightarrow{w}$ is contained in $\overleftrightarrow{v}$. Assign thus to each node $v \in V \setminus Y$ values $\overleftarrow{v}$ and $\overrightarrow{v}$, and consider set $N = \{\overleftrightarrow{v} \mid v \in V \setminus Y\}$ of *node intervals* and set $Q = \{[sl(\overleftarrow{v})..sl(\overrightarrow{v})] \, : \, v \in V \setminus Y\}$ of *query intervals*.

LEMMA 8.16 *Let v and w be two nodes in V such that $sl(v) = w$. Then, $\overleftrightarrow{w}$ is the smallest interval in N that contains $[sl(\overleftarrow{v})..sl(\overrightarrow{v})]$. In other words, w is the* lowest common ancestor (LCA) *of leaves $sl(\overleftarrow{v})$ and $sl(\overrightarrow{v})$ in ST_T.*

Proof Assume that node $u \in V \setminus Y$ is such that interval $\overleftrightarrow{u}$ includes $[sl(\overleftarrow{v})..sl(\overrightarrow{v})]$ and $\overleftrightarrow{u}$ is smaller than $\overleftrightarrow{w}$. Then u must be a descendant of w, so $\ell(u) = \ell(w) \cdot X$, where $\ell(v) = a \cdot \ell(w)$, $a \in \Sigma$, and $|X| > 0$. It follows that $T[\mathsf{SA}[sl(\overleftarrow{v})]..\mathsf{SA}[sl(\overleftarrow{v})] + |\ell(w)| + |X| - 1] = T[\mathsf{SA}[sl(\overrightarrow{v})]..\mathsf{SA}[sl(\overrightarrow{v})] + |\ell(w)| + |X| - 1] = \ell(w)X$, and thus $T[\mathsf{SA}[\overleftarrow{v}]..\mathsf{SA}[\overleftarrow{v}] + |\ell(w)| + |X|] = T[\mathsf{SA}[\overrightarrow{v}]..\mathsf{SA}[\overrightarrow{v}] + |\ell(w)| + |X|] = a\ell(w)X$. Hence v would have a single outgoing edge prefixed by X, which amounts to a contradiction. □

We thus need to find, for each interval in Q, the smallest interval in N that includes it. For this we can use a general result on *batched lowest common ancestor* queries.

LEMMA 8.17 *Let $\mathcal{T} = (V, E)$ be a tree on n leaves numbered $1, 2, \ldots, n$. Given a set of query intervals $Q = \{[i..j] \mid 1 \leq i \leq j \leq n\}$ of size $O(n)$, one can link each $q \in Q$ to its lowest common ancestor $v \in V$ in $O(n)$ time.*

Proof During a left-to-right depth-first traversal of $\mathcal{T}$ we open an annotated parenthesis "$(_v$" when we first visit a node v, and we close an annotated parenthesis "$)_v$" when we last visit a node v. This produces a valid parenthesization (see Figure 8.6, top).

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

Assume that, when we visit a leaf i, we first print "$[_q$" for each query $q \in Q$ such that $\overleftarrow{q} = i$, and then we print "$]_q$" for each $q \in Q$ such that $\overrightarrow{q} = i$. Assume that the resulting parenthesization is stored in a doubly-linked list, with additional links from each "$)_v$" to the corresponding "$(_v$", from each "$]_q$" to the corresponding "$[_q$", and from each "$[_q$" to the closest parenthesis of type "(" on its left. Use call `prev`$[q]$ to denote this link, which is initially set to the immediate predecessor of "$[_q$" in the linked list if it is of type "(", and otherwise remains undefined. Then, scan the parenthesization from left to right. If we meet a closing parenthesis "$)_v$" we delete it, we reach the corresponding "$(_v$", and we delete it as well. If we meet a closing parenthesis "$]_q$", we delete it, we reach the corresponding "$[_q$", we delete it as well, and we start a right-to-left iteration by following links in `prev` to locate the closest opening parenthesis of type "(" to the left of "$[_q$". If `prev`$[q']$ points to a "$(_w$" for some "$[_{q'}$" encountered during this iteration, we stop and we set `prev`$[q'']$ to point to "$(_w$" as well for every "$[_{q''}$" that has been traversed so far. Note that the list before "$[_q$" contains only opening parentheses "$(_w$" for which no corresponding closing parenthesis "$)_w$" has yet been found, therefore $\overleftrightarrow{w}$ is necessarily the smallest interval that contains q and it is safe to create the link q to w. Clearly, at the end of the process, all links to lowest common ancestors from query intervals are created. The size of the parenthesization is $O(n)$, and processing each parenthesis takes *amortized* constant time: indeed, after following k links while searching for the closest parenthesis of type "(", we assign $k - 1$ `prev` values, and each such `prev` value is assigned exactly once. □

Figure 8.6 (bottom) shows an instantiation of Lemma 8.17 on a suffix tree and on query intervals $Q = \{[sl(\overleftarrow{v})..sl(\overrightarrow{v})] \ : \ v \in V \setminus Y\}$. The following theorem immediately follows.

THEOREM 8.18 *Assume we are given* ST_T *and the suffix links from all its leaves. Then, the suffix links from all internal nodes of* ST_T *can be built in* $O(n)$ *time.*

8.4 Applications of the suffix tree

In this section we sketch some prototypical applications of suffix trees. The purpose is to show the versatility and power of this fundamental data structure. The exercises of this chapter illustrate some additional uses, and later chapters revisit the same problems with more space-efficient solutions.

In what follows, we often consider the suffix tree of a set of strings instead of just one string. Then, a property of a set $\mathcal{S} = \{S^1, S^2, \ldots, S^d\}$ should be interpreted as a property of the concatenation $C = S^1\$_1S^2\$_2 \cdots \$_{d-1}S^d\#$, where characters $\$_i$ and # are all distinct and do not appear in the strings of $\mathcal{S}$.

8.4.1 Maximal repeats

A *repeat* in a string $T = t_1t_2 \cdots t_n$ is a substring X that occurs more than once in T. A repeat X is *right-maximal* (respectively, *left-maximal*) if it cannot be extended to the

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

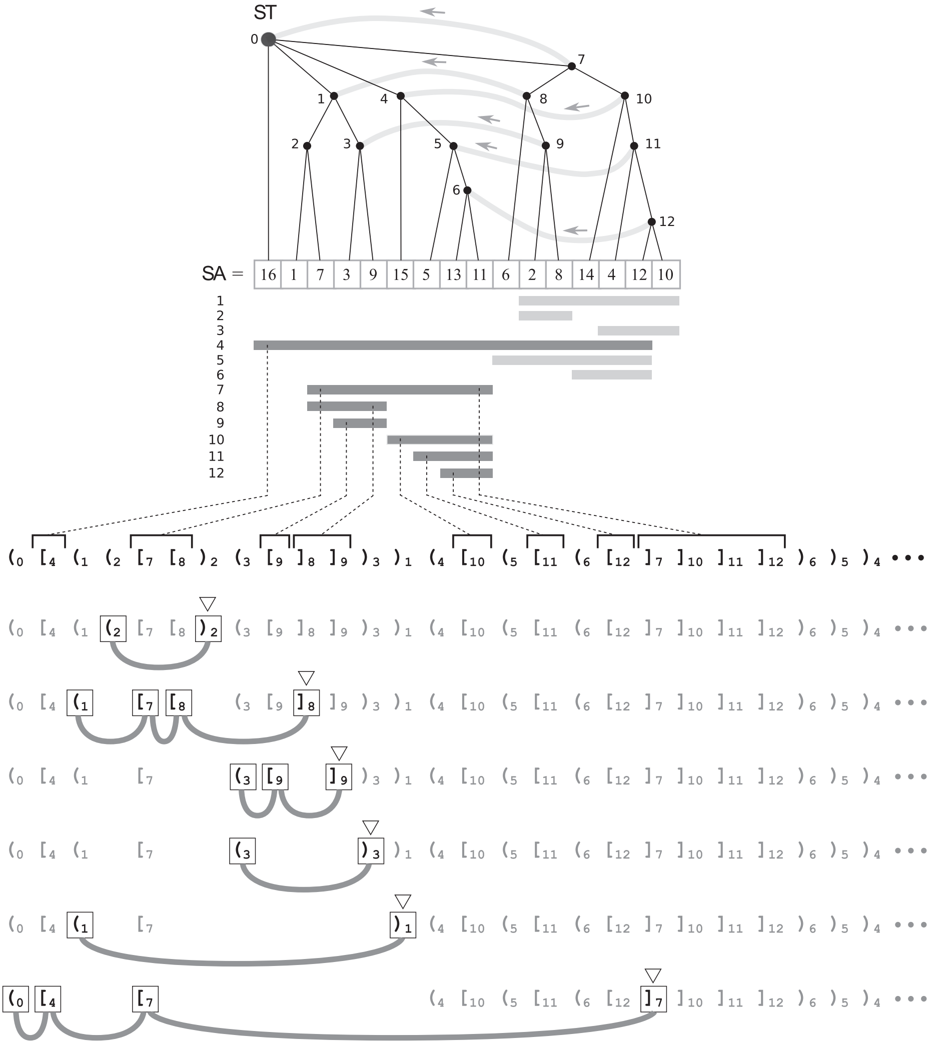
ST
0
1
2
3
4
5
6
7
8
9
10
11
12
SA = 16 1 7 3 9 15 5 13 11 6 2 8 14 4 12 10
1
2
3
4
5
6
7
8
9
10
11
12
$(_0\ [_4\ (_1\ (_2\ [_7\ [_8\ )_2\ (_3\ [_9\ ]_8\ ]_9\ )_3\ )_1\ (_4\ [_{10}\ (_5\ [_{11}\ (_6\ [_{12}\ ]_7\ ]_{10}\ ]_{11}\ ]_{12}\ )_6\ )_5\ )_4\ \cdots$

Figure 8.6 Building the suffix links of internal nodes from the suffix links of leaves. (Top) The intervals $[sl(i_v)..sl(j_v)]$ for every internal node v (gray bars), where i_v and j_v are the first and the last position of the interval of v in SA, and the parenthesization produced by a depth-first traversal of ST. For clarity, character labels are omitted and nodes are assigned numerical identifiers. (Bottom) Left-to-right scan of the parenthesization, represented as a doubly-linked list with shortcuts connecting open and closed parentheses. Triangles show the current position of the pointer.

right (respectively, to the left), by even a single character, without losing at least one of its occurrences. A repeat is *maximal* if it is both left- and right-maximal. For example, the maximal repeats of $T =$ ACAGCAGT are A and CAG.

Maximal repeat detection (Problem 8.2) is tightly connected to the properties of the suffix tree of T.

Problem 8.2 Maximal repeats

Find all maximal repeats in a string T, that is, all substrings of T that are both left- and right-maximal.

It suffices to build $\mathsf{ST}_T = (V, E)$ and to traverse V. Indeed, note that the right-maximal substrings of T are exactly the labels of the internal nodes of ST_T, thus the maximal repeats of T correspond to a subset of $V \setminus Y$, where Y is the set of leaves of ST_T. To test the left-maximality of an internal node, it suffices to perform a linear-time post-order traversal of ST_T, keeping an integer $\texttt{previous}[u] \in \Sigma \cup \{\#\}$ at each node u. Assume that all children of an internal node u have already been explored, and that each such child v has $\texttt{previous}[v] = \#$ if V is the leaf that corresponds to suffix $T[1..n]$, or if there are at least two leaves in the subtree of v whose corresponding suffixes $T[i..n]$ and $T[j..n]$ in T are such that $T[i-1] \neq T[j-1]$. Otherwise, $\texttt{previous}[v] = c \in \Sigma$, where c is the character that precedes all suffixes of T that correspond to leaves in the subtree of v. Then, we set $\texttt{previous}[u] = \#$ if there is a child v of u with $\texttt{previous}[v] = \#$, or if there are at least two children v and w of u such that $\texttt{previous}[v] \neq \texttt{previous}[w]$. Otherwise, we set $\texttt{previous}[u] = \texttt{previous}[v]$ for some child v of u. The maximal repeats of T are then the labels of all internal nodes u with $\texttt{previous}[u] = \#$, and their occurrences in T are stored in the leaves that belong to the subtree rooted at u. By Theorem 8.15 we thus have the following fact.

THEOREM 8.19 *Problem 8.2 of listing all maximal repeats of a string $T = t_1 t_2 \cdots t_n \in [1..\sigma]^*$ and their occurrences can be solved in $O(\sigma + n)$ time.*

Maximal repeats can also be characterized through the suffix-link tree SLT_T: recall Figure 8.4. Note that every leaf of SLT_T either has more than one Weiner link or corresponds to $T\#$. The set of all leaves of SLT_T (excluding $T\#$) and of all its internal nodes with at least two (implicit or explicit) Weiner links coincides with the set of all substrings of T that are maximal repeats. The set of all left-maximal substrings of T enjoys the *prefix closure* property, in the sense that, if a string is left-maximal, so is any of its prefixes. It follows that the maximal repeats of T form a subtree of the suffix tree of T rooted at r (thick black lines in Figure 8.4).

Figure 8.7 visualizes concepts that are derived from maximal repeats: a *supermaximal repeat* is a maximal repeat that is not a substring of another maximal repeat, and a *near-supermaximal repeat* is a maximal repeat that has at least one occurrence that is not contained inside an occurrence of another maximal repeat. Maximal exact matches will be defined in Section 11.1.3 and maximal unique matches will be our next topic.

8.4.2 Maximal unique matches

A *maximal unique match (MUM)* of two string S and T is a substring X that occurs exactly once in both S and T and that cannot be extended to the left or to the right without losing one of the occurrences. That is, for a MUM X it holds that $X = s_i s_{i+1} \cdots s_{i+k-1}$,

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

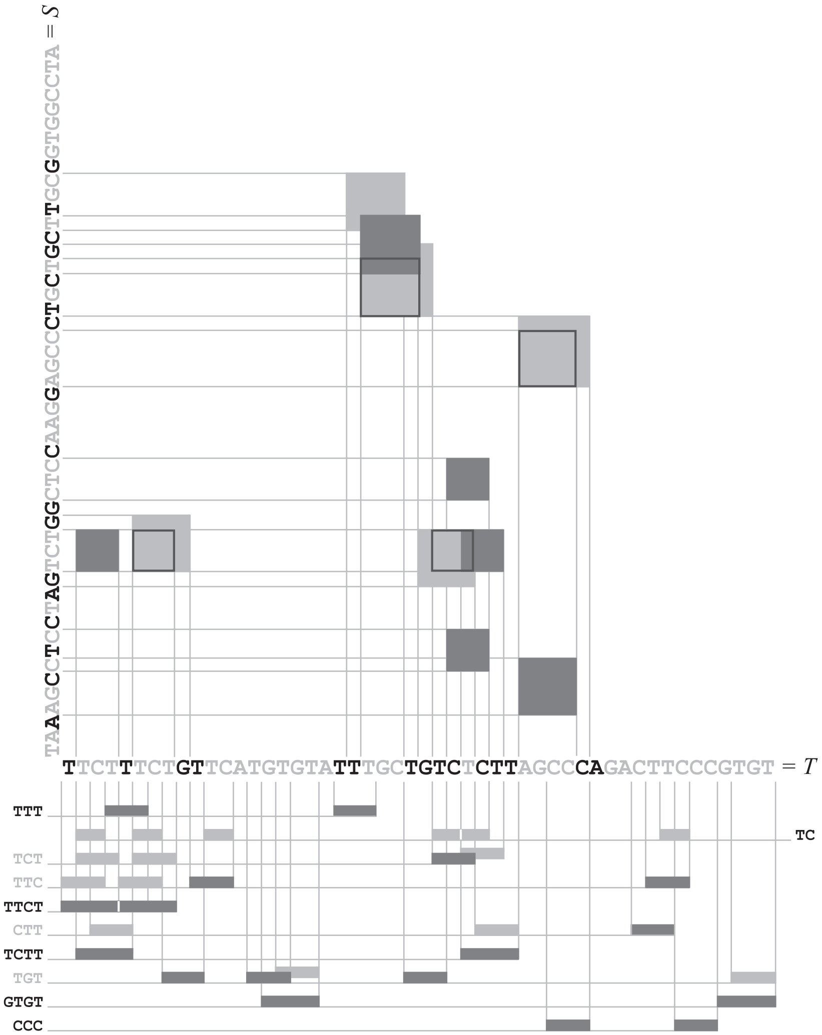
TAAGCCTCCTAGTCTGGCTCCAAGGAGCCCTGCTGCTTGCGGTGGCCTA = S
TTCTTTCTGTTCATGTGTATTTGCTGTCTCTTAGCCCAGACTTCCCGTGT = T
TTT
TC
TCT
TTC
TTCT
CTT
TCTT
TGT
GTGT
CCC

Figure 8.7 (Top) Examples of maximal exact matches (dark-gray rectangles) and maximal unique matches (light-gray rectangles) between string S and string T. Empty rectangles with black border show pairs of occurrences of AGCC, TGCT, and TCT that are not maximal exact matches, because at least one of their flanking characters matches in S and T. (Bottom) Examples of supermaximal repeats (black strings) and of near-supermaximal repeats (gray strings) of string T, and their occurrences (rectangles). The occurrences of a repeat that are not covered by the occurrences of any other repeat of T are highlighted in dark gray. Note that maximal repeat TC is not near-supermaximal, since all its occurrences are covered by occurrences of TTC or TCT.

$X = t_j t_{j+1} \cdots t_{j+k-1}$, $s_{i-1} \neq t_{j-1}$, and $s_{i+k} \neq t_{j+k}$ (for clarity, we omit the boundary cases here). To detect all the MUMs of two strings S and T, it suffices again to build the suffix tree $\mathsf{ST}_C = (V, E)$ of the concatenation $C = S\$T\#$ and to traverse V. Indeed, MUMs are right-maximal substrings of C, so, like maximal repeats, they are a subset of $V \setminus Y$. Moreover, only internal nodes $v \in V$ with exactly two leaves as children can be MUMs. Let the two leaves of a node v be associated with suffixes $C[i..|C|]$ and

$C[j..|C|]$, respectively. Then, i and j must be such that $i \leq |S|$ and $j > |S+1|$, and the left-maximality of v can be checked by accessing $S[i-1]$ and $T[j-1]$ in constant time.

The notion of maximal unique matches extends naturally to a set of strings, as stated in Problem 8.3.

Problem 8.3 Maximal unique matches

Given a set of strings $\mathcal{S} = \{S^1, S^2, \ldots, S^d\}$ of total length $n = \sum_{i=1}^{d} |S^i|$, report all maximal repeats X of $\mathcal{S}$ that are *maximal unique matches*, that is, X appears exactly once in each string of $\mathcal{S}$.

To solve Problem 8.3, we can build again the generalized suffix tree $\mathsf{ST}_C = (V, E)$ of the concatenation $C = S^1\$_1S^2\$_2 \cdots \$_{d-1}S^d\#$. We can determine left-maximal nodes with the same post-order traversal of ST_C we used for maximal repeats, and we can detect nodes $v \in V$ with exactly d leaves in their subtree. However, we need also to check that no two leaves in the subtree of v come from the same string of $\mathcal{S}$. This can be done by filling a bitvector $I[1..d]$ (initialized to zeros), setting at each leaf i under v value $I[k] = 1$ if suffix $C_{\mathsf{SA}[i]..|C|}$ starts inside string S^k. Value k can be stored at leaf i during the construction of the suffix tree. After filling I under node v, if all bits are set to 1, each of the d suffixes under v must originate from a different string, and hence maximal node v defines a maximal unique match. This process can be repeated for each maximal node v having exactly d leaves under it. It follows that the checking of d bits at node v can be amortized to the leaves under it, since each leaf corresponds to only one v to which this process is applied. By Theorem 8.15 we thus have the following result.

THEOREM 8.20 *Problem 8.3 of finding all the maximal unique matches (MUMs) of a set of strings of total length n can be solved in $O(\sigma + n)$ time.*

8.4.3 Document counting

A generalization of the MUM problem is to report in how many strings a (right-) maximal repeat appears, as stated in Problem 8.4.

Problem 8.4 Document counting

Given a set of strings $\mathcal{S} = \{S^1, S^2, \ldots, S^d\}$ of total length $n = \sum_{i=1}^{d} |S^i|$, compute for each (right-)maximal repeat X of $\mathcal{S}$ the number of strings in $\mathcal{S}$ that contain X as a substring.

To solve Problem 8.4, we can start exactly as in the solution for MUMs by building the generalized suffix tree $\mathsf{ST}_C = (V, E)$ of the concatenation $C = S^1\$_1S^2\$_2 \cdots \$_{d-1}S^d\#$. For a node $v \in V$, let $n(v, k)$ be the number of leaves in the subtree rooted at v that

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

correspond to suffixes of C that start inside string S^k. Recall the algorithm for building the suffix tree of a string T from its suffix array and LCP array (Figures 8.5(b) and (c)). We now follow the same algorithm, but add some auxiliary computations to its execution. Assign to each internal node v the temporary counter $\texttt{leafCount}[v]$, storing the total number of leaves in the subtree rooted at v, and the temporary counter $\texttt{duplicateCount}[v] = \sum_{i=1}^{d} \max\{0, n(v,k) - 1\}$. At the end of the process, we make sure that these temporary counters obtain their final values. The solution to Problem 8.4 is given by $\texttt{leafCount}[v] - \texttt{duplicateCount}[v]$ and available for each internal node v, corresponding thus to a right-maximal repeat. Observe that the solution to the MUMs computation is a special case with $\texttt{leafCount}[v] = d$ and $\texttt{duplicateCount}[v] = 0$ for nodes v that also correspond to left-maximal repeats.

To start the update process, counters $\texttt{leafCount}[v]$ and $\texttt{duplicateCount}[v]$ are initially set to zero for each v. At each step of the process, $\texttt{leafCount}[v]$ stores the number of leaves in the current subtree rooted at any node v, except for nodes in the *right-most path* of the current suffix tree, for which $\texttt{leafCount}[v]$ stores the number of leaves in the current subtree of v, minus the number of leaves in the subtree rooted at the *right-most child* of v. Let us call this property the *leaf count invariant*. When we insert a new leaf below an internal node w, it suffices to increment $\texttt{leafCount}[w]$ by one, and when we visit a node v while traversing the right-most path bottom-up, it suffices to increment $\texttt{leafCount}[u]$ by $\texttt{leafCount}[v]$, where u is the parent of v. These updates guarantee that the leaf count invariant will hold.

We can compute $\texttt{duplicateCount}[v]$ by reusing the batched lowest common ancestor computation from Lemma 8.17. Assume that the set of query intervals Q is empty at the beginning. Let $\texttt{document}[1..|C|]$ be a vector that stores at $\texttt{document}[i]$ value k if $\mathsf{SA}_C[i]$ starts inside string S^k in C. Building this array is left to Exercise 8.20. Assume that we are scanning SA_C from left to right, and that we are currently at position i. Our strategy consists in incrementing $\texttt{duplicateCount}[v]$ for the node v that is the lowest common ancestor of two closest leaves i and j of ST_C such that $\texttt{document}[i] = \texttt{document}[j]$, that is, $\texttt{document}[k] \neq \texttt{document}[i]$ for all $k \in [i+1..j-1]$. Observe that this process is like *dueling* i and j, since one of them is announced as a duplicate at v. Specifically, assume that we are currently processing leaf i. We maintain for each document identifier $k \in [1..d]$ a variable $\texttt{previous}[k]$ that stores the largest $j \in [1..i-1]$ such that $\texttt{document}[j] = k$. Then we can access $j = \texttt{previous}[\texttt{document}[i]]$ at leaf i. Once we have updated the current suffix tree using suffix $\mathsf{SA}_C[i]$, we set $\texttt{previous}[\texttt{document}[i]] = i$ and we insert $[j..i]$ into the set of query intervals Q. After processing all leaves, we apply Lemma 8.17 on ST_C and set Q. As a result, we obtain for each query interval $q \in Q$ the lowest common ancestor v in ST_C. For each pair (q, v) we increment $\texttt{duplicateCount}[v]$ by one.

Finally, we *propagate* the values of $\texttt{duplicateCount}$ (increment the parent counter by the child counters) from such lowest common ancestors to all nodes of ST_C by a linear-time traversal of the tree. This guarantees that all counter values will be correctly computed since a duplicate leaf under node v is a duplicate leaf under each ancestor of v. By Theorem 8.15 we thus have the following result.

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

THEOREM 8.21 *Problem 8.4 of document counting on a set of strings of total length n can be solved in $O(\sigma + n)$ time.*

8.4.4 Suffix–prefix overlaps

In later chapters we will present variants of the *fragment assembly* problem, which consists in connecting together substrings of an unknown string (typically short DNA fragments, known as *short reads*) on the basis of their shared prefixes and suffixes. As a preview, we consider here the following problem.

Problem 8.5 All-against-all suffix–prefix overlaps

Given a set of strings $\mathcal{R} = \{R^1, R^2, \ldots, R^d\}$ of total length n and a threshold τ, output all pairs (R^i, R^j) such that $R^i = XY$ and $R^j = YZ$ for some X, Y, Z in Σ^*, where $|Y| \geq \tau$.

Problem 8.5 can be solved again with a suitable depth-first traversal of a suffix tree. Indeed, build the generalized suffix tree ST_T of $T = R^1\$_1R^2\$_2 \cdots \$_{d-1}R^d\$_d\#$, and imagine that an empty stack is allocated to every string in $\mathcal{R}$. Assume that, during a depth-first traversal of ST_T, we encounter for the first time a node v with string depth at least τ: we push its string depth to the stacks of all R^i such that there is an edge from v starting with character $\$_i$. Symmetrically, when v is visited for the last time, we pop from all the corresponding stacks. Clearly, when we reach a leaf node corresponding to suffix $R^i\$_i \cdots$ of T, the non-empty stacks in memory are exactly those that correspond to strings R^j having a suffix of length at least τ that equals a prefix of R^i, and the top of the non-empty stack j contains the length of the longest suffix of R^j that equals a prefix of R^i.

We should keep in memory only the non-empty stacks at each step of the traversal. This can be done using a doubly-linked list, to which we append in constant time a new stack, and we delete in constant time an empty stack. An auxiliary table $P[1..d]$ points to the (variable) position in the linked list of the stack associated with each string in $\mathcal{R}$. The following result is thus immediately obtained.

THEOREM 8.22 *Problem 8.5 of all-against-all suffix–prefix overlaps can be solved in $O(\sigma + n + \texttt{pairs})$ time, where* `pairs` *is the number of pairs in the given set of string $\mathcal{R}$ that overlap by at least τ characters.*

8.5 Literature

We just scratched the surface of k-mer indexes, giving almost folklore solutions. Elias coding is from Elias (1975). The literature on inverted files (see for example Baeza-Yates & Ribeiro-Neto (2011)) contains more advanced techniques that can also be

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

adapted to *k*-mer indexes. Suffix arrays were first introduced in Gonnet *et al.* (1992) and Manber & Myers (1993), and our doubling algorithm for their construction mimics the one in Manber & Myers (1993). The linear-time construction algorithm presented here is from Ko & Aluru (2005). Simultaneously and independently, two other linear-time construction algorithms were invented (Kärkkäinen *et al.* 2006, Kim *et al.* 2005): all of these follow similar divide-and-conquer recursive strategies, but differ considerably in technical details. We chose to present Ko's & Aluru's solution because it contains as a sub-result a useful technique for sorting sets of strings. By encapsulating this sub-result in a conceptual black box, the rest of the algorithm can be presented in relatively self-contained modules. Nevertheless, as with other linear-time suffix array construction algorithms, the very interplay among all steps is key to achieving linear time: such deep relationships make these algorithms one of the most fascinating achievements in the field of string processing. For readers wishing to obtain a more detailed exposition on the different suffix array construction algorithms, there is an excellent survey available by Puglisi *et al.* (2007).

Suffix trees are an earlier invention: the 24th Annual Symposium on Combinatorial Pattern Matching (CPM 2013) celebrated the 40th anniversary of Weiner's foundational paper (Weiner 1973). Construction algorithms with different properties have been considered (McCreight 1976; Ukkonen 1995; Farach 1997). We propose an exercise that sketches the online construction (Ukkonen 1995). Many textbooks describe this algorithm in detail, because it is the simplest to understand and very likely the most educative. We chose to give the readers of this book the chance to "rediscover" some of its parts. The first linear-time suffix tree construction algorithm on large alphabets was given in Farach (1997): this algorithm can indirectly construct suffix arrays in linear time, and it already uses a divide-and-conquer recursion strategy similar to the later direct suffix array constructions mentioned above. Owing to the existence of efficient and practical suffix array construction algorithms, we chose to give the details of just the folklore construction of the suffix tree from the suffix array and LCP array. The surprisingly simple construction of the LCP array (left as an exercise) is from Kasai *et al.* (2001).

For learning more advanced properties of suffix trees and their variants, such as suffix automata and compact suffix automata, we refer the interested reader to stringology textbooks like Crochemore & Rytter (2002). We mentioned just a few illustrative applications of suffix trees to biological sequence analysis, whose connections to the rest of the book will become clear in the forthcoming chapters: the maximal unique matches computation is folklore and hinted to us by Gonzalo Navarro, document counting is from Hui (1992), and the overlap computation is from Gusfield *et al.* (1992). More related applications are described by Gusfield (1997), whose presentation we partly followed to introduce the repeat finding concepts. We made a significant conceptual simplification by introducing *batched lowest common ancestor queries*, which we also use for the construction of suffix links. Usually these queries are solved with rather advanced data structures for *range minimum queries* (Bender & Farach-Colton 2000; Harel & Tarjan 1984; Fischer & Heun 2011), but we showed that one can collect all such queries and then use a simple scan to collect all answers at once.

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

Other theoretically important text indexing data structures we did not discuss are the *directed acyclic word graph* (DAWG) and its compacted version, the CDAWG: these are covered by Exercises 8.24 and 8.25. The DAWG was introduced in Blumer *et al.* (1985), and the CDAWG in Crochemore & Vérin (1997b). Efficient construction algorithms are presented in Apostolico & Lonardi (2002) and Crochemore & Vérin (1997a). The relationship between CDAWG and maximal repeats described in Exercise 8.25 was studied in Raffinot (2001).

Exercises

8.1 Consider a k-mer index in which the pointers to the lists of occurrences of k-mers are encoded using a table of size σ^k in which the ith element is a pointer to the list of the occurrence positions of k-mer $P = p_1p_2\cdots p_k$ in $T = t_1t_2\cdots t_n$ for $i = p_1 \cdot \sigma^{k-1} + p_2 \cdot \sigma^{k-2} + \cdots + p_k$, where each $p_j \in \{0, 1, \ldots, \sigma - 1\}$. Show that both this table and the gamma-encoded lists can be constructed in $O(\sigma^k + n)$ time by two scans from left to right over T. *Hint.* Use a first scan to determine the size of the lists and a second scan to fill both the lists and the table.

8.2 The table of the previous assignment is sparse if k is large. Show that by using techniques from Section 3.2 one can represent the table in $\sigma^k(1 + o(1)) + \ell \log(n + 2n \log \ell)$ bits, where ℓ is the number of non-empty lists, while still retaining constant-time access to the list. *Hint.* Use a bitvector marking the non-empty elements and supporting rank queries. Concatenate all encodings of the occurrence lists and store a pointer to the starting position of each list inside the concatenation.

8.3 Show how to represent the pointer table of the previous assignment in $\sigma^k(1 + o(1)) + (\ell + n(1 + 2\log \ell))(1 + o(1))$ bits. *Hint.* Use a bitvector supporting rank queries and another supporting select queries.

8.4 Derive an analog of Theorem 8.2 replacing γ-coding with δ-coding. Do you obtain a better bound?

8.5 The order-k *de Bruijn graph* on text $T = t_1t_2\cdots t_n$ is a graph whose vertices are $(k - 1)$-mers occurring in T and whose arcs are k-mers connecting $(k - 1)$-mers consecutive in T. More precisely, let $\texttt{label}(v) = \alpha_1\alpha_2\cdots\alpha_{k-1}$ denote the $(k - 1)$-mer of vertex v and $\texttt{label}(w) = \beta_1\beta_2\cdots\beta_{k-1}$ the $(k - 1)$-mer of vertex w. There is an arc e from vertex v to w with $\texttt{label}(e) = \alpha_1\alpha_2\cdots\alpha_{k-1}\beta_{k-1}$ iff $\alpha_2\alpha_3\cdots\alpha_{k-1} = \beta_1\beta_2\cdots\beta_{k-2}$ and $\texttt{label}(e)$ is a substring of T. See Section 9.7 for more details on de Bruijn graphs and on their efficient representation. Modify the solution to Exercise 8.1 to construct the order-k de Bruijn graph.

8.6 The table of size σ^k on large k makes the above approach infeasible in practice. Show how hashing can be used for lowering the space requirement.

8.7 Another way to avoid the need for a table of size σ^k is to use a suffix array (enhanced with an LCP array) or a suffix tree. In the suffix tree of T, consider all paths from the root that spell strings of length k; these are the distinct k-mers occurring in T. Consider one such path labeled by X ending at an edge leading to node v. The leaves

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

in the subtree of v contain all the suffixes prefixed by k-mer X in lexicographic order. Show that one can sort all these lexicographically sorted lists of occurrences of k-mers in $O(n)$ time, to form the k-mer index.

8.8 Modify the approach in the previous assignment to construct the de Bruijn graph in linear time.

8.9 Visualize the suffix array of the string `ACGACTGACT#` and simulate a binary search on the pattern `ACT`.

8.10 Recall the indicator vector $I[1..n]$ for string $T = t_1 t_2 \cdots t_n$ with $I[i] = 1$ if suffix $T_{i..n}$ is small, that is, $T_{i..n} < T_{i+1..n}$, and $I[i] = 0$ otherwise. Show that it can be filled with one $O(n)$ time scan from right to left.

8.11 Recall R and SA_R from the linear-time suffix array construction. Give pseudocode for mapping SA_R to suffix array SA_T of T so that $\mathsf{SA}_T[i]$ is correctly computed for small suffixes.

8.12 Give the pseudocode for the linear-time algorithm sketched in the main text to fill suffix array entries for large suffixes, assuming that the entries for small suffixes have already been computed.

8.13 We assumed there are fewer small suffixes than large suffixes. Consider the other case, especially, how does the string sorting procedure need to be changed in order to work correctly for substrings induced by large suffixes? Solve also the previous assignments switching the roles of small and large suffixes.

8.14 Show how to compute the $\mathsf{LCP}[2..n]$ array in linear time given the suffix array. *Hint.* With the help of the inverse of the suffix array, you can scan the text $T = t_1 t_2 \cdots t_n$ from left to right, comparing suffix $t_1 t_2 \cdots$ with its predecessor in suffix array order, suffix $t_2 \cdots$ with its predecessor in suffix array order, and so on. Observe that the common prefix length from the previous step can be used in the next step.

8.15 Visualize the suffix tree of a string `ACGACTGACT` and mark the nodes corresponding to maximal repeats. Visualize also Weiner links.

8.16 Consider the suffix tree built from a suffix array. Give a linear-time algorithm to compute suffix links for its leaves.

8.17 Visualize the suffix tree on a concatenation of two strings of your choice and mark the nodes corresponding to maximal unique matches. Choose an example with multiple nodes with two leaf children, of which some correspond to maximal unique matches, and some do not.

8.18 Show that the number of implicit Weiner links can be bounded by $2n - 2$. *Hint.* Show that all implicit Weiner links can be associated with unique edges of the suffix tree of the reverse.

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

8.19 Show that the number of implicit Weiner links can be bounded by n. *Hint.* Characterize the edges where implicit Weiner links are associated in the reverse. Observe that these edges form a subset of a cut of the tree.

8.20 Recall the indicator vector $I[1..|C|]$ in the computation of MUMs on multiple sequences. Show how it can be filled in linear time.

8.21 The *suffix trie* is a variant of suffix tree, where the suffix tree edges are replaced by unary paths with a single character labeling each edge: the concatenation of unary path labels equals the corresponding suffix tree edge label. The suffix trie can hence be quadratic in the size of the text (which is a reason for the use of suffix trees instead). Now consider the suffix link definition for the nodes of the suffix trie that do *not* correspond to nodes of the suffix tree. We call such nodes *implicit nodes* and such suffix links *implicit suffix links* of the suffix tree. Show how to *simulate* an implicit suffix link $sl(v,k) = (w,k')$, where $k > 0$ ($k' \geq 0$) is the position inside the edge from the parent of v to v (respectively, parent of w to w), where $v = (v,0)$ and $w = (w,0)$ are *explicit* nodes of the suffix tree and (v,k) (possibly also (w,k')) is an implicit node of the suffix tree. In the simulation, you may use parent pointers and explicit suffix links. What is the worst-case computation time for the simulation?

8.22 Consider the following *descending suffix walk* by string $S = s_1s_2\cdots s_m$ on the suffix *trie* of $T = t_1t_2\cdots t_n$. Walk down the suffix tree of T as deep as the prefix of S still matches the path followed. When you cannot walk any more, say at s_i, follow a suffix link, and continue walking down as deep as the prefix of $s_is_{i+1}\cdots$ still matches the path followed, and follow another suffix link when needed. Continue this until you reach the end of S. Observe that the node v visited during the walk whose string depth is greatest defines the *longest common substring* of S and T: the string X labeling the path from root to v is such a substring. Now, consider the same algorithm on the suffix *tree* of T. Use the simulation of implicit suffix links from the previous assignment. Show that the running time is $O(\sigma + |T| + |S| \log \sigma)$, i.e. the time spent in total for simulating implicit suffix links can be *amortized* on scanning S.

8.23 The suffix tree can also be constructed directly without requiring the suffix array. *Online* suffix tree construction works as follows. Assume you have the suffix tree $\mathsf{ST}(i)$ of $t_1t_2\cdots t_i$, then update it into the suffix tree $\mathsf{ST}(i+1)$ of $t_1t_2\cdots t_{i+1}$ by adding t_{i+1} at the end of each suffix. Let $(v_1,k_1),(v_2,k_2),\ldots,(v_i,k_i)$ be the (implicit) nodes of $\mathsf{ST}(i)$ corresponding to paths labeled $t_1t_2\cdots t_i$, $t_2t_3\cdots t_i$, ..., t_i, respectively (see the assigments above for the notion of implicit and explicit nodes). Notice that $sl(v_j,k_j) = (v_{j+1},k_{j+1})$.

(a) Show that $(v_1,k_1),(v_2,k_2),\ldots,(v_i,k_i)$ can be split into lists $(v_1,0),\ldots,(v_l,0)$, $(v_{l+1},k_{l+1}),\ldots,(v_a,k_a)$, and $(v_{a+1},k_{a+1}),\ldots,(v_i,k_i)$ such that nodes in the first list are leaves before and after appending t_{i+1}, nodes in the second list do not yet have a branch (or a next character along the edge) starting with character t_{i+1}, and nodes in the third list already have a branch (or a next character along the edge) starting with character t_{i+1}.

https://doi.org/10.1017/CBO9781139940023.010 Published online by Cambridge University Press

(b) From the above it follows that the status of nodes $(v_1, 0), \ldots, (v_l, 0)$ remains the same and no updates to their content are required (the start position of the suffix is a sufficient identifier). For $(v_{l+1}, k_{l+1}), \ldots, (v_a, k_a)$ a new leaf for t_{i+1} needs to be created. In the case of explicit node $(v, 0)$, this new leaf is inserted under node v. In the case of implicit node (v, k), the edge from the parent of v to v is split after position k, and a new internal node is created, inheriting v as child and the old parent of v as parent, and t_{i+1} as a new leaf child. The suffix links from the newly created nodes need to be created during the traversal of the list $(v_{l+1}, k_{l+1}), \ldots, (v_{a+1}, k_{a+1})$. Once one has detected (v_{a+1}, k_{a+1}) being already followed by the required character and created the suffix link from the last created node to the correct successor of (v_{a+1}, k_{a+1}), say (w, k'), the list does not need to be scanned further. To insert t_{i+2}, one can start from the new *active point* (w, k') identically as above, following suffix links until one can walk down with t_{i+2}. Notice the analogy to a descending suffix walk, and use a similar analysis to show that the simulation of implicit suffix links amortizes and that the online construction requires $O(n)$ steps (the actual running time is multiplied by an alphabet factor depending on how the branching is implemented).

8.24 The *directed acyclic word graph* (DAWG) is an automaton obtained by minimizing a suffix trie. Suppose that we build the suffix trie of a string $T\#$. Then two nodes v and u of the suffix trie will be merged if the substring that labels the path to v is a suffix of the substring that labels the path to u and the frequencies of the two substrings in $T\#$ are the same. Show that the number of vertices in the DAWG is linear in the length of T, by showing that the following statements hold.

(a) There is a bijection between the nodes of the suffix-link tree of $\underline{T}\#$ and the vertices of the DAWG labeled by strings whose frequency is at least two in $T\#$.

(b) There is a bijection between all prefixes of $T\#$ that have exactly one occurrence in $T\#$ and all the vertices of the DAWG labeled by strings whose frequency is exactly one in $T\#$.

Then show that the number of edges is also linear, by showing a partition of the edges into three categories:

(a) edges that are in bijection with the edges of the suffix-link tree of $\underline{T}\#$,

(b) edges that are in bijection with a subset of the edges of the suffix tree of $T\#$, and

(c) edges that connect two nodes that are in bijection with two prefixes of $T\#$ of lengths differing by one.

8.25 The *compacted DAWG* (CDAWG) of a string $T\#$ is obtained by merging every vertex of the DAWG of $T\#$ that has just one outgoing arc with the destination of that arc. Equivalently, the CDAWG can be obtained by minimizing the suffix tree of T in the same way as the suffix trie is minimized to obtain the DAWG. Note that the resulting automaton contains exactly one vertex with no outgoing arc, called the *sink*. Show that there is a bijection between the set of maximal repeated substrings of $T\#$ and the set of vertices of the CDAWG, excluding the root and the sink.