BWT	t	c	a	$\$$	a	t	c	a	a	a	a
$B^{\$}$	0	0	0	**1**	0	0	0	0	0	0	0
B^a	0	0	**1**	0	**1**	0	0	**1**	**1**	**1**	**1**
B^c	0	**1**	0	0	0	0	**1**	0	0	0	0
B^t	**1**	0	0	0	0	**1**	0	0	0	0	0

Figure 7.12: Indicator bit vectors of BWT $= tca\$atcaaaa$.

7.3 Backward search

Ferragina and Manzini [100] showed that it is possible to search a pattern $P = P[1..m]$ backwards in the suffix array SA of string S, without storing SA. A backward search means that we first search for the $P[m]$-interval, then for the $P[m-1..m]$-interval, and so on, until the whole pattern $P[1..m]$ is found. In the computer science literature, any data structure that allows to search a pattern P backwards in the (conceptual) suffix array of a string S is called an ***FM-index*** of S. Before showing how a backward search works, we introduce a simple FM-index consisting of the C-array and certain indicator bit vectors. In Section 7.4 we will become acquainted with another FM-index: the wavelet tree.

7.3.1 A simple FM-index

Definition 7.3.1 Given a string (text) T of length n on the alphabet Σ,

- $rank_c(T, i)$ returns the number of occurrences of character $c \in \Sigma$ in the prefix $T[1..i]$,
- $select_c(T, i)$ returns the position of the i-th occurrences of character $c \in \Sigma$ in T.

It what follows, we are interested in data structures that support these kinds of queries efficiently. Since we are mainly interested in the Burrows-Wheeler transform of a string S, we fix $T =$ BWT. However, the techniques developed below work for arbitrary strings T.

The easiest method to support $rank_c(\mathsf{BWT}, i)$ and $select_c(\mathsf{BWT}, i)$ queries is to use σ many indicator bit vectors of length n. For each character $c \in \Sigma$, the bit vector B^c is defined by $B^c[i] = 1$ if and only if $\mathsf{BWT}[i] = c$; see Figure 7.12. Clearly, $rank_c(\mathsf{BWT}, i) = rank_1(B^c, i)$ and $select_c(\mathsf{BWT}, i) = select_1(B^c, i)$. Therefore, the problem is reduced to the problem of answering rank and select queries on bit vectors. This can be done in constant time with a total of $n\sigma + o(n\sigma)$ bits of space.

Given the ability to answer $rank_c(\mathsf{BWT}, i)$ and $select_c(\mathsf{BWT}, i)$ queries in constant time, it is possible to compute $LF(i)$ and $\psi(i)$ in constant time as well. This can be seen as follows. According to Definition 7.2.3, if

F	\$	a	a	a	a	a	a	c	c	t	t
B_F	**1**	**1**	0	0	0	0	0	**1**	0	**1**	0

Figure 7.13: The bit vector B_F for $F = \$aaaaaacctt$.

$\mathsf{BWT}[i] = c$ is the k-th occurrence of character c in the BWT-array, then $LF(i) = j$ is the index so that $F[j]$ is the k-th occurrence of c in the array F. Furthermore, we have seen that the k-th occurrence of c in F can be found at index $C[c] + k$. It follows as a consequence that

$$LF(i) = C[c] + rank_c(\mathsf{BWT}, i), \text{ where } c = \mathsf{BWT}[i] \tag{7.1}$$

Note that in the computer science literature, $Occ(c, i)$ is often used instead of $rank_c(\mathsf{BWT}, i)$.

Because the ψ-function is the inverse of the LF-mapping (Lemma 7.2.9), it follows that if $F[i] = c$ is the k-th occurrence of c in the array F, then $\psi(i) = j$ is the index so that $\mathsf{BWT}[j] = c$ is the k-th occurrence of character c in the BWT-array. So once we know c and k, $\psi(i) = j$ can be obtained by $j = select_c(\mathsf{BWT}, k)$. In fact, it is sufficient to know c because $k = i - C[c]$. Clearly, c can be obtained from F because $F[i] = c$. However, storing F would be a waste of memory because we can use the array C instead. By doing a binary search on C, we can determine in $O(\log \sigma)$ time the character c with $C[c] < i \leq C[c+1]$, i.e., $c = \max\{a \in \Sigma \mid C[a] < i\}$. Alternatively, c can be determined in constant time with a $rank$ data structure on the bit vector B_F defined by $B_F[1] = 1$ and, for all l with $2 \leq l \leq n$, $B_F[l] = 1$ if and only if $F[l-1] \neq F[l]$; see Figure 7.13 for an example. This is because $c = \Sigma[m]$, where $m = rank_1(B_F, i)$. All in all, we have

$$\psi(i) = select_c(\mathsf{BWT}, i - C[c]), \text{ where } c = \Sigma[rank_1(B_F, i)] \tag{7.2}$$

Note that the array C can be completely replaced with the bit vector B_F because $C[c] = select_1(B_F, m) - 1$.

In summary, if we use the indicator bit vectors and the bit vector B_F, then $LF(i)$ and $\psi(i)$ can be computed in constant time, using $n(1+\sigma) + o(n(1+\sigma))$ bits of space. If we use the C-array instead of the bit vector B_F, then $LF(i)$ can also be computed in constant time, but the computation of $\psi(i)$ takes $O(\log \sigma)$ time. In this case, $\sigma \log n + n\sigma + o(n\sigma)$ bits of space are required.

Exercise 7.3.2 Give a linear-time algorithm that takes the string BWT as input and returns the bit vector B_F.

7.3.2 The search algorithm

Let us return to the issue of backward search. As already mentioned, a backward search means that we first search for the $P[m]$-interval, then for

i	BWT	$S_{\mathsf{SA}[i]}$
1	t	\$
$\rightarrow 2$	c	$aaacatat\$$
3	a	$aacatat\$$
4	\$	$acaaacatat\$$
5	a	$acatat\$$
6	t	$at\$$
$\rightarrow 7$	c	$atat\$$
8	a	$caaacatat\$$
9	a	$catat\$$
10	a	$t\$$
11	a	$tat\$$

i	BWT	$S_{\mathsf{SA}[i]}$
1	t	\$
$\rightarrow 2$	c	$aaacatat\$$
$\rightarrow 3$	a	$aacatat\$$
4	\$	$acaaacatat\$$
5	a	$acatat\$$
6	t	$at\$$
7	c	$atat\$$
8	a	$caaacatat\$$
9	a	$catat\$$
10	a	$t\$$
11	a	$tat\$$

Figure 7.14: Searching pattern aa backwards in $S = acaaacatat\$$. Given the a-interval $[2..7]$, one backward search step determines the aa-interval $[i..j]$ by $i = C[a] + rank_a(\mathsf{BWT}, 2-1) + 1 = 1 + 0 + 1 = 2$ and $j = C[a] + rank_a(\mathsf{BWT}, 7) = 1 + 2 = 3$.

the $P[m-1..m]$-interval, and so on, until the whole pattern $P[1..m]$ is found. For example, the a-interval in the suffix array of the string $S = acaaacatat\$$ is $[2..7]$; see Figure 7.14. That is, $S_{\mathsf{SA}[2]}, S_{\mathsf{SA}[3]}, \ldots, S_{\mathsf{SA}[7]}$ are the only suffixes in S that start with an a. Consequently, if we search for the suffixes starting with aa, then $S_{\mathsf{SA}[2]-1}, S_{\mathsf{SA}[3]-1}, \ldots, S_{\mathsf{SA}[7]-1}$ are the sole candidates because only these suffixes have an a at the second position. Note that these candidates can be found in the suffix array at $LF(2), LF(3), \ldots, LF(7)$. Out of these candidates only those that have an a at first position belong to the aa-interval. Because $S[\mathsf{SA}[i] - 1] = a$ if and only if $\mathsf{BWT}[i] = a$, the suffix $S_{\mathsf{SA}[i]-1}$ at index $LF(i)$ belongs to the aa-interval if and only if $S_{\mathsf{SA}[i]}$ belongs to the a-interval and $\mathsf{BWT}[i] = a$. As a matter of fact, it suffices to know the first index p and the last index q with $2 \le p \le q \le 7$ and $\mathsf{BWT}[p] = a = \mathsf{BWT}[q]$. (This is because the suffixes $S_{\mathsf{SA}[2]}, S_{\mathsf{SA}[3]}, \ldots, S_{\mathsf{SA}[7]}$ are ordered lexicographically and if one prepends the same character to all of them, then the resulting strings will occur in the same lexicographic order.) In our example, we have $p = 3$ and $q = 5$. Hence the boundaries of the aa-interval are $LF(3) = 2$ and $LF(5) = 3$. The crucial question is how to find p and q efficiently. Observe that a linear scan of the BWT array would result in a bad worst-case running time. In fact, we do not have to know p and q, as we shall see below.

Suppose in general that we know the ω-interval $[i..j]$ of some suffix ω of P, say $\omega = P[b..m]$. Next, we have to determine the $c\omega$-interval, where

Algorithm 7.7 Given an ω-interval $[i..j]$ and a character c, this procedure returns the $c\omega$-interval if it exists; otherwise, it returns $\perp$.

$backwardSearch(c, [i..j])$
    $i \leftarrow C[c] + rank_c(\mathsf{BWT}, i-1) + 1$
    $j \leftarrow C[c] + rank_c(\mathsf{BWT}, j)$
  if $i \leq j$ **then**
    return interval $[i..j]$
  else
    return $\perp$

$c = P[b-1]$. Assume for a moment that the $c\omega$-interval is non-empty, i.e., $c\omega$ is a substring of S. Let p and q be the smallest and largest index with $i \leq p \leq q \leq j$ and $\mathsf{BWT}[p] = c = \mathsf{BWT}[q]$. As discussed above, the $c\omega$-interval is the interval $[LF(p)..LF(q)]$. According to Equation 7.1 (page 300) we have

$$\begin{aligned} LF(p) &= C[c] + rank_c(\mathsf{BWT}, p) \\ &= C[c] + rank_c(\mathsf{BWT}, p-1) + 1 \\ &= C[c] + rank_c(\mathsf{BWT}, i-1) + 1 \end{aligned}$$

where the last equality follows from the fact that p is the index of the first occurrence of c in $\mathsf{BWT}[i..j]$. Analogously,

$$\begin{aligned} LF(q) &= C[c] + rank_c(\mathsf{BWT}, q) \\ &= C[c] + rank_c(\mathsf{BWT}, j) \end{aligned}$$

because q is the index of the last occurrence of c in $\mathsf{BWT}[i..j]$. We conclude that the $c\omega$-interval $[C[c] + rank_c(\mathsf{BWT}, i-1) + 1..C[c] + rank_c(\mathsf{BWT}, j)]$ can be determined without knowing p and q. Pseudo-code for one backward search step can be found in Algorithm 7.7. In the preceding discussion, we assumed that the $c\omega$-interval is non-empty. What happens if it is empty? Then, $rank_c(\mathsf{BWT}, i-1) = rank_c(\mathsf{BWT}, j)$. This implies that $C[c] + rank_c(\mathsf{BWT}, i-1) + 1 > C[c] + rank_c(\mathsf{BWT}, j)$ and thus Algorithm 7.7 returns the undefined value $\perp$.

Pseudo-code for searching the whole pattern P is given in Algorithm 7.8.

Exercise 7.3.3 Show that backward search can be accomplished in $O(m \log n)$ time, solely based on the ψ-array of S (cf. Definition 7.2.7).

Algorithm 7.8 Given a pattern P, this procedure returns the P-interval if it exists; otherwise, it returns $\perp$.

```
backwardSearch(P)
  i ← 1
  j ← n
  k ← m
  while i ≤ j and k ≥ 1 do
    c ← P[k]
    i ← C[c] + rank_c(BWT, i − 1) + 1
    j ← C[c] + rank_c(BWT, j)
    k ← k − 1
  if i ≤ j then
    return interval [i..j]
  else
    return ⊥
```

7.4 Wavelet trees

The *wavelet tree* was introduced by Grossi et al. [134]. In a very general sense, a wavelet tree is a binary tree[4] that has exactly σ many leaves and there is a bijection between the set of leaves and Σ (i.e., each of the leaves corresponds to a distinct character from the alphabet Σ). Moreover, every internal node v stores a bit vector B^v equipped with rank and select data structures.

The conceptually easiest way to introduce wavelet trees goes as follows. We say that an interval $[l..r]$ is an *alphabet interval* if it is a subinterval of $[1..\sigma]$, where $\sigma = |\Sigma|$. For an alphabet interval $[l..r]$, the string $\mathsf{BWT}^{[l..r]}$ is obtained from the Burrows-Wheeler transformed string BWT of S by deleting all characters in BWT that do not belong to the subalphabet $\Sigma[l..r]$ of $\Sigma[1..\sigma]$. As an example, consider the string $\mathsf{BWT} = tca\$atcaaaa$ and the alphabet interval $[1..2]$. The string $\mathsf{BWT}^{[1..2]}$ is obtained from $tca\$atcaaaa$ by deleting the characters c and t. Thus, $\mathsf{BWT}^{[1..2]} = a\$aaaaa$. Each node v of the tree corresponds to a string $\mathsf{BWT}^{[l..r]}$, where $[l..r]$ is an alphabet interval. The root of the tree corresponds to the string $\mathsf{BWT} = \mathsf{BWT}^{[1..\sigma]}$. If $l = r$, then v has no children. Otherwise, v has two children: its left child v_L corresponds to the string $\mathsf{BWT}^{[l..m]}$ and its right child v_R corresponds to the string $\mathsf{BWT}^{[m+1..r]}$, where $m = \lfloor \frac{l+r}{2} \rfloor$. In this case, v stores a bit vector, denoted by B^v or $B^{[l..r]}$, whose i-th entry is 0 if the i-th character in $\mathsf{BWT}^{[l..r]}$ belongs to the subalphabet $\Sigma[l..m]$ and 1 if it belongs to the subalphabet $\Sigma[m+1..r]$. To put it differently, an entry in the bit vector is 0 if the corresponding character belongs to the left subtree and 1 if it

[4] That is, every node in the tree is either a leaf or has exactly two children.