2

How the Burrows-Wheeler Transform works

This chapter will look in detail at how the Burrows-Wheeler Transform is implemented in practice. The examples given in Chapter 1 overlooked some important practical details — to transform a text of n characters the encoder was sorting an array of n strings, each n characters long, and the decoder performed n sorts to reverse the transform. This complexity is not necessary for the BWT, and in this chapter we will see how to perform the encoding and decoding in $O(n)$ space, and $O(n \log n)$ time. In fact, using a few tricks, the time can be reduced to $O(n)$.

We will also look at various auxiliary data structures that are used for decoding the Burrows-Wheeler Transform, as some of them, while not essential for decoding, are useful if the transformed text is to be searched. These extra structures can still be constructed in $O(n)$ time so in principle they add little to the decoding cost.

This chapter considers only the transform; in the next chapter we will look at how a compression system can take advantage of the transformed text to reduce its size; we refer to this second phase as the "Local to Global Transform", or LGT.

We will present the Burrows-Wheeler Transform for coding a string T of n characters, $T[1 \ldots n]$, over an alphabet Σ of $|\Sigma|$ characters. Note that there is a summary of all the main notation in Appendix A on page 309.

2.1 The forward Burrows-Wheeler Transform

The forward transform essentially involves sorting all rotations of the input string, which clusters together characters that occur in similar contexts. Figure 2.1a shows the rotations A that would occur if the transform is given T = `mississippi` as the input[1], and Figure 2.1b shows the result of sorting A, which we will refer to as A_s.

[1] We will use `mississippi` as a running example in this chapter. This string is often used in the literature as an example because it illustrates the features of

mississippi	imississipp
ississippim	ippimississ
ssissippimi	issippimiss
sissippimis	ississippim
issippimiss	mississippi
ssippimissi	pimississip
sippimissis	ppimississi
ippimississ	sippimissis
ppimississi	sissippimis
pimississip	ssippimissi
imississipp	ssissippimi
(a)	(b)

Fig. 2.1. (a) The array A containing all rotations of the input `mississippi`; (b) A_s, obtained by sorting A. The last column of A_s (usually referred to as L) is the Burrows-Wheeler Transform of the input

However, rather than use $O(n^2)$ space as suggested by Figure 2.1, we can create an array $R[1 \ldots n]$ of references to the rotated strings in the input text T. Initially $R[i]$ is simply set to i for each i from 1 to n, as shown in Figure 2.2a, to represent the unsorted list. It is then sorted using the substring beginning at $T[R[i]]$ as the comparison key. Figure 2.2b shows the result of sorting; for example, position 11 is the first rotated string in lexical order (`imiss...`), followed by position 8 (`ippim...`) and position 5 (`issip...`); the final reference string is $R = [11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]$.

The array R directly indexes the characters in T corresponding to the first column of A_s, referred to as F in the BWT literature. The last column of A_s (referred to as L) is the output of the BWT, and can be read off as $T[R[i]-1]$, where i ranges from 1 to n (if the index to T is 0 then it refers to $T[n]$). In this case the transformed text is $L =$ `pssmipissii`. We also need to transmit an index a to indicate to the decoder which position in L corresponds to the last character of the original text (i.e. which row of A_s contains the original string T). In this case the index $a = 5$ is included.

In the above description the transform is completed using just $O(n)$ space (for R). The time taken is $O(n)$ for the creation of the array R , plus the time needed for sorting. Conventionally sorting is regarded as taking $O(n \log n)$ average time if a standard method such as quicksort is used. However, some string sequences can cause near-worst-case behavior in some versions of quicksort, particularly if there is a lot of repetition in the string and the pivot for quicksort is not selected carefully. This corresponds to the traditional $O(n^2)$ worst-case of quicksort where the data is already sorted — if T contains long runs of the same character then the A array will contain long sorted sequences.

---

the BWT well. Note that there is no unique sentinel (end of string) symbol in this example; it is not essential for the BWT, although it can simplify some aspects, particularly when we deal with suffixes later.

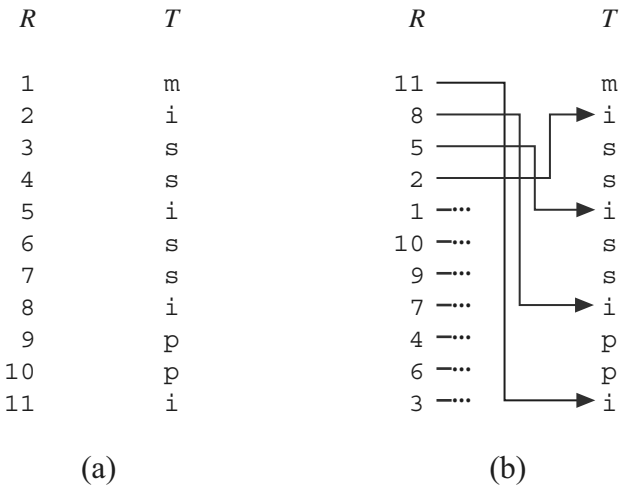

Fig. 2.2. The R array used to sort the sample file `mississippi`

For example, Figure 2.3 shows the A array for the input `aaaaaab`. It is already sorted because of the way the `b` terminates the long sequence of `a` characters. It is possible to avoid this worst case behavior in quicksort with techniques such as the median-of-three partition selection, but the nature of the BWT problem means that even better sorting methods are possible.

Not only can the pre-sorted list cause poor performance in some versions of quicksort, but the long nearly identical prefixes mean that lexical comparisons will require many character comparisons, which means that the constant-time assumption for comparisons is invalid; if all the characters are identical then it could take $O(n)$ time for each of the $O(n^2)$ comparisons, which would be extremely slow, especially considering that for such a case the BWT involves no permutations at all. Long repeated strings can occur in practice in images that contain many pixels of the same color (such as a scan of a black-and-white page with little writing on it) and in genomic data where the alphabet is very small and repeated substrings are common.

```
aaaaaab
aaaaaba
aaaabaa
aaabaaa
aabaaaa
abaaaaa
baaaaaa
```

Fig. 2.3. The array A containing all rotations of the input `aaaaaab`

There are several ways to avoid this problem. Burrows and Wheeler observed in their original paper that by having a unique sentinel character, the sorting problem is equivalent to sorting all the suffixes in T, which can be done in linear time and space using a suffix tree. This is discussed in more detail in Chapters 4 and 4, but we should mention that the main drawback of this approach is that although the space requirement is $O(n)$, the constant factor can be significant.

Instead, Burrows and Wheeler proposed a modified version of quicksort that applies a radix sort to the first two characters of each sort key. Each of the two-character buckets now needs to be sorted, but special attention is paid to buckets where the first two characters are the same, since these are likely to indicate long runs of the same character (typically null or space characters), which can take a long time to make a lexical comparison for comparison based sorting, yet are trivial to sort because of how they were generated. Eventually quicksort is only applied to groups of substrings that need sorting within buckets. For example, the strings in Figure 2.3 would be split into three buckets for those beginning with `aa`, `ab` and `bb` respectively. The `aa` bucket does not automatically have quicksort applied to it because the first two characters are the same, and indeed in this case the bucket happens to already be sorted, and would cause long comparisons between strings because of the long prefixes of runs of the letter `a`.

Another approach is to eliminate this problem by coding long runs of the same character using a run-length encoding technique, where runs of repeated characters are replaced with a shorter code. This can sometimes even have a positive effect on the amount of compression, although the main purpose is to avoid the poor sorting speed that occurs in the special cases described above by eliminating long runs of the same character. One downside of this is that the original text is no longer available directly in the BWT, which can affect some of the compressed-domain searching methods described later in this book. Also, the run-length encoding will change the context information that the BWT uses, hence the effect on compression is not necessarily positive.

One issue that is inevitable with the BWT is that it requires a large block of memory to store the input string (T) and the index to the strings being sorted (R). If the block is too small the compression will be poor, but if too large, it may use too much memory. Even if the memory is available, there can be issues with caching, and there are performance benefits from keeping blocks within the size of a cache, not just within main memory. The pattern of access to the memory will be random because of the sorting operations that need to be done (the same problem occurs during decoding as well). On modern computers there can be several layers of caching that will be trying to guess the memory access patterns, and these may have complex interactions with the accesses needed for the BWT. This concern needs to be taken into consideration when deciding on the block size; if it fits within the cache (and not just within main memory), it may well be able to operate faster. On the other hand, as parallel machines with on-board memory become more popular

the BWT method can potentially be adapted to take advantage of this kind of architecture, and it is even possible that it will have performance benefits in a parallel environment over other popular compression methods. The actual performance in practice will depend on the architecture of the machine, the amount of memory available, and the design of any caches.

Appendix B lists web sites that provide a variety of source code for performing the BWT. Some are suitable for experimenting with the transform and tracing the process, while others are production systems the have optimized the details of coding to perform well in practice.

2.2 The reverse Burrows-Wheeler Transform

The reverse transform — taking a BWT permuted text and reconstructing the original input T — is somewhat more difficult to implement than the forward transform, but it can still be done in $O(n)$ time and space if care is taken. The example given in Figure 1.2 reconstructed the A_s array, but as for encoding, in practice there is no need to store this $O(n^2)$ array. Generally two $O(n)$ index arrays will be needed, plus two $O(|\Sigma|)$ arrays to count the characters in the input. There are several ways that decoding can be done. The original paper by Burrows and Wheeler produces the output in reverse, although it is not difficult to produce the output in the original order. We will show how to generate data structures for both of these cases.

We will use the decoding of the string `mississippi` as a running example. Figure 2.4 shows the array A_s for this example, with columns F and L labeled. A_s is not stored explicitly in practice, but we shall use it in the meantime to illustrate how decoding can be done. The decoder can determine F simply by sorting L, since it contains exactly the same characters, just in a different order — each column of A_s contains the same set of characters because the rows are all the rotations of the original string. In fact, F need not be stored, as it can be generated implicitly by counting how often each character appears in L.

Looking at A_s helps us to see the information that is needed to perform the decoding. Given just F and L, the key step is determining which character should come after a particular character in F. Consider, for example, the two rows ending with a `p` (rows 1 and 6). Because of the rotation, the order of these two rows is determined by the characters that come after the respective occurrences of `p` in T (`imi...` and `pim...` respectively). Thus the first occurrence of `p` in L corresponds to the first occurrence of `p` in F, and likewise with the second occurrence. This enables us to work through the text backwards: if we have just decoded the second `p` in L, then it must correspond to the one in row 7 of F. Looking at row 7, the L column tells us that the `p` was preceded by an `i`. In turn, because this is the second `i` in L, it must correspond to the second `i` in F, which is in row 2. We carry on traversing the L and F arrays in this way until the whole string is decoded — in reverse.

	F										L
1	i	m	i	s	s	i	s	s	i	p	p
2	i	p	p	i	m	i	s	s	i	s	s
3	i	s	s	i	p	p	i	m	i	s	s
4	i	s	s	i	s	s	i	p	p	i	m
5	m	i	s	s	i	s	s	i	p	p	i
6	p	i	m	i	s	s	i	s	s	i	p
7	p	p	i	m	i	s	s	i	s	s	i
8	s	i	p	p	i	m	i	s	s	i	s
9	s	i	s	s	i	p	p	i	m	i	s
10	s	s	i	p	p	i	m	i	s	s	i
11	s	s	i	s	s	i	p	p	i	m	i

Fig. 2.4. The array A_s for `mississipi`; F and L are the first and last columns respectively

The correspondence could also have been used to decode the string in its original order. For example, looking at the `p` in $L[6]$, we can determine that it is followed by $F[6]$, a `p`. Since this is the first `p` in F, it corresponds to the first `p` in L, that is, row 1. That `p` is followed by an `i`, and so on. It is marginally simpler to decode the string in reverse order, so usually the BWT literature uses the backwards decoding, although we shall be using both orders in this book.

An easy way to follow the above relationships is to number the appearances of the characters in F and L. Figure 2.5 shows the F and L columns from Figure 2.4, but we have numbered the occurrences of each character in order from first to last using subscripts. This makes the decoded string easy to read off; for example, the fourth row has $L[4] = \mathtt{m}_1$, and the corresponding $F[4]$ tells us that it is followed by $\mathtt{i}_4$. Since $\mathtt{i}_4$ is in $L[11]$, we can get the next character from $F[11]$, which is $\mathtt{s}_4$. The entire string is decoded in the order $\mathtt{m}_1\mathtt{i}_4\mathtt{s}_4\mathtt{s}_2\mathtt{i}_3\mathtt{s}_3\mathtt{s}_1\mathtt{i}_2\mathtt{p}_2\mathtt{p}_1\mathtt{i}_1$.

In practice the decoder never reconstructs A_s or F in full, but implicitly creates indexes to represent enough of its structure to decode the original string. L is stored explicitly (the decoder just reads the input and stores it in L), but F is stored implicitly to save space and to efficiently provide the kind of information needed during decoding.

Figure 2.6 shows three auxiliary arrays that are useful for decoding. $K[c]$ is simply a count of how many times each character c occurs in F, which is easily determined by counting the characters in L. $M[c]$ locates the first position of character c in the array F, so K and M together effectively store the information in F. $C[i]$ stores the number of times the character $L[i]$ occurs in L earlier than position i; for example, the last character in L is `i`, and `i`

	F	L
1	$\mathtt{i}_1$	$\mathtt{p}_1$
2	$\mathtt{i}_2$	$\mathtt{s}_1$
3	$\mathtt{i}_3$	$\mathtt{s}_2$
4	$\mathtt{i}_4$	$\mathtt{m}_1$
5	$\mathtt{m}_1$	$\mathtt{i}_1$
6	$\mathtt{p}_1$	$\mathtt{p}_2$
7	$\mathtt{p}_2$	$\mathtt{i}_2$
8	$\mathtt{s}_1$	$\mathtt{s}_3$
9	$\mathtt{s}_2$	$\mathtt{s}_4$
10	$\mathtt{s}_3$	$\mathtt{i}_3$
11	$\mathtt{s}_4$	$\mathtt{i}_4$

Fig. 2.5. Using the character order to perform the reverse transform

occurs 3 times in the earlier part of L. These three arrays make it easy to traverse the input in reverse.

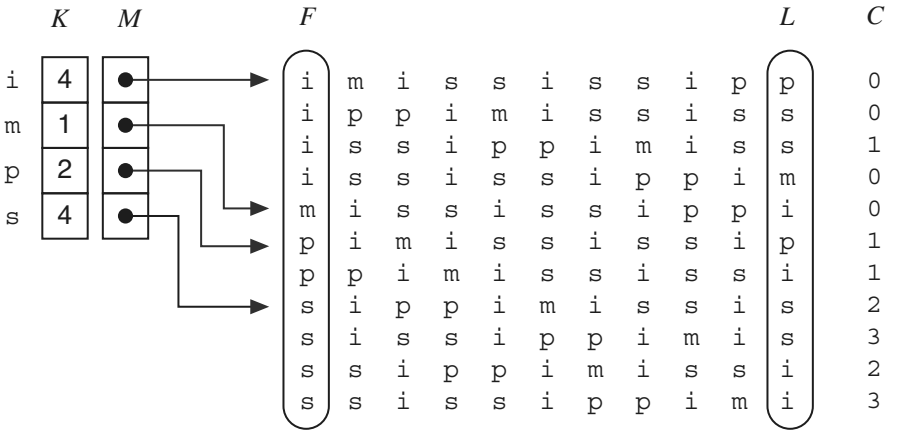

Fig. 2.6. The array (A_s) that is implicitly reconstructed to decode the string `pssmipissii`

Algorithm 2.1 shows how the input (transformed text L and starting index a) is used to construct these three arrays, which are then used to produce Q, the decoded text. The first step is simply to count the characters into K by going through L (the input), shown in lines 1 to 7 of the algorithm. At the same time, it is convenient to construct C by recording the value of K before each increment. The array M is then constructed in lines 10 to 14 by accumulating the values in C. We now have sufficient structures to decode the text in reverse, which happens in lines 16 to 20.

```
BWT-Decode(L, a)
 1 for c ← 1 to |Σ| do
 2    K[c] ← 0
 3 end for
 4
 5 for i ← 1 to n do
 6    C[i] ← K[L[i]]
 7    K[L[i]] ← K[L[i]] + 1
 8 end for
 9
10 sum ← 1
11 for c ← 1 to |Σ| do
12    M[c] ← sum
13    sum ← sum + K[c]
14 end for
15
16 i ← a
17 for j ← n downto 1 do
18    Q[j] ← L[i]
19    i ← C[i] + M[L[i]]
20 end for
```

Algorithm 2.1: Reconstruction of the original text

Note that the decoder needs to be given the index a, which is the element in L that corresponds to the end of the text. In our running example a would be 5 — it corresponds to the row in A_s that represents the original string T. $L[5]$ gives us `i` as the last character of Q. The corresponding value in C is 0, which means that this is the first occurrence of an `i` in L. Thus it corresponds to the first occurrence of `i` in F, which is found by adding $C[i]$ to $M[L[i]]$, in line 19 of the algorithm. This proceeds until n characters have been decoded, at which point the whole string is stored in Q.

Reversing the BWT this way requires four arrays (L, K, C and M). K and M contain just $|\Sigma|$ entries (the characters are represented by integers from 1 to $|\Sigma|$) and are likely to be of negligible size; L and C contain n values, and hence use $O(n)$ space. We would normally also have to allow for Q, which uses $O(n)$ space to store the backwards string before it can be stored in the correct order. The time taken is also $O(n) + O(|\Sigma|)$, since the main work is in the two passes through the n input items — once to count them, and once to decode them.

It may be inconvenient that the output is generated backwards, and there are two ways to address this. Below we will look at how to use an extra auxiliary array to do this, but if the goal is simply to decode the input, this is less efficient than using the temporary Q array to store the output. An even simpler approach is to reverse the order of the string at encoding time.

This should not take any extra time, since the whole string must be read into memory anyway — we simply fill the array T in reverse. It may have an impact on compression, depending on the type of data, but for most data there will be no significant impact, and it may even improve compression. The cases where there is an impact tend to be binary files with specific patterns such as leading zeros before numbers. In such cases it is worth being aware of the issue anyway, and the ordering should be chosen to suit the data, since the amount of compression can depend on details such as whether the representation puts the most significant byte of a large number first or last (big- or little-endian). For textual data, reversing the input string means that the system is based around prior contexts of characters, which is how many other compression methods work anyway.

If the transformed text is to be decoded multiple times, it is possible to store one or more auxiliary arrays that enable us to traverse sections of the text at will. This can be useful for pattern matching because it allows segments of the original string to be read off when needed for matching, but still relates the data to the implicit sorted array A_s, which provides access to a sorted list of strings that are useful for searching.

The value $C[i] + M[L[i]]$ is the key to navigating through L to decode the original string, so instead of doing the decoding immediately (which was in lines 16 to 20 of Algorithm 2.1), an array V is created to store the navigation information, shown in Algorithm 2.2. This array can then be used to step backwards through the original characters; the character at $L[i]$ is preceded by the character at $L[V[i]]$. The values of V for our running example are shown in Figure 2.7.

```
COMPUTE-ARRAY-V(C, M, L) 1  i ← a
2  for j ← n downto 1 do
3      V[i] ← C[i] + M[L[i]]
4      i ← V[i]
5  end for
```

Algorithm 2.2: Creating the array V to allow for efficient future decoding of the input

It is just as easy to create an auxiliary array that will decode the original text forwards rather than backwards. This array will be called W, and it identifies the position of the character in L that comes *after* the present one, compared with V, which gives the position that comes *before*. As for V, this new array is not essential for decoding, but it can be useful because it preserves access to the sorted structure of L, which can be exploited during pattern matching. Figure 2.7 shows the values of W for the running example.

Algorithm 2.3 shows how the W array can be created. Note that the array M that was created in Algorithm 2.1 is used, and that afterwards its contents

	F										*L*	*V*	*W*
1	i	m	i	s	s	i	s	s	i	p	p	6	5
2	i	p	p	i	m	i	s	s	i	s	s	8	7
3	i	s	s	i	p	p	i	m	i	s	s	9	10
4	i	s	s	i	s	s	i	p	p	i	m	5	11
5	m	i	s	s	i	s	s	i	p	p	i	1	4
6	p	i	m	i	s	s	i	s	s	i	p	7	1
7	p	p	i	m	i	s	s	i	s	s	i	2	6
8	s	i	p	p	i	m	i	s	s	i	s	10	2
9	s	i	s	s	i	p	p	i	m	i	s	11	3
10	s	s	i	p	p	i	m	i	s	s	i	3	8
11	s	s	i	s	s	i	p	p	i	m	i	4	9

Fig. 2.7. The auxiliary arrays V and W which can be used to decode the sample string

are changed so they are no longer valid. Like V, the W array is created in just $O(n)$ time.

COMPUTE-ARRAY-W(M, L)
1 **for** $i \leftarrow 1$ **to** n **do**
2 $\quad W[M[L[i]]] \leftarrow i$
3 $\quad M[L[i]] \leftarrow M[L[i]] + 1$
4 **end for**

Algorithm 2.3: Creating the array W to allow for future decoding of the input

W can then be used to generate the original text in its correct order using the simple sequence shown in Algorithm 2.4.

DECODE-WITH-ARRAY-W(W, L)
1 $i \leftarrow a$
2 **for** $j \leftarrow 1$ **to** n **do**
3 $\quad Q[j] \leftarrow L[i]$
4 $\quad i \leftarrow W[i]$
5 **end for**

Algorithm 2.4: Decoding the original text in its correct order using W

If both forwards and backwards generation of the original text is needed, it is possible to create V and W in one pass as shown in Algorithm 2.5.

V and W are essentially a mapping between F and L in each direction. In some of the pattern matching algorithms in Chapter 7 we will also recreate the array R that was used in the encoder to store the sort order of the substrings, and the reverse mapping of R, called R'. These provide a mapping between F and T; for example, if $R[i]$ is k, then $F[i]$ was the k-th character in T, and $R'[k]$ will be i. All of the arrays for our `mississippi` example are shown in Figure 2.8, and the algorithm for creating R and R' from W is given in Algorithm 2.6.

i	T	F	L	C	V	W	R'	R
1	m	i	p	0	6	5	5	11
2	i	i	s	0	8	7	4	8
3	s	i	s	1	9	10	11	5
4	s	i	m	0	5	11	9	2
5	i	m	i	0	1	4	3	1
6	s	p	p	1	7	1	10	10
7	s	p	i	1	2	6	8	9
8	i	s	s	2	10	2	2	7
9	p	s	s	3	11	3	7	4
10	p	s	i	2	3	8	6	6
11	i	s	i	3	4	9	1	3

Fig. 2.8. Array values that can be used to do the BWT and searching of the text `mississippi`

```
COMPUTE-ARRAYS-V-AND-W(M, L)
1 for i ← 1 to n do
2     V[i] ← M[L[i]]
3     W[M[L[i]]] ← i
4     M[L[i]] ← M[L[i]] + 1
5 end for
```

Algorithm 2.5: Creating both the V and W arrays in one pass

2.3 Special cases

In the previous examples the auxiliary arrays traverse each character in L to recreate the original text. There is a special case for the BWT that occurs if the

```
COMPUTE-ARRAYS-R-AND-R'(W)
1 i ← a
2 for j ← 1 to n do
3     R'[j] ← i
4     R[i] ← j
5     i ← W[i]
6 end for
```

Algorithm 2.6: Construction of R and R' auxiliary arrays in the decoder

input text T is nothing but repetitions of a substring, such as `blahblahblah`, or even `aaaaaaa`. If this happens, some of the rotations of the text will be identical, and the reverse transform will end up using only one of the substring occurrences for decoding.

For example, the text `cancan` results in the decoding arrays shown in Figure 2.9. The arrows show the cycle of three characters that will occur following the V for W links; the other three characters in L are never used. This will still decode correctly; it is just that it is important to decode n times, rather than relying on coming back to the starting point to determine when to stop.

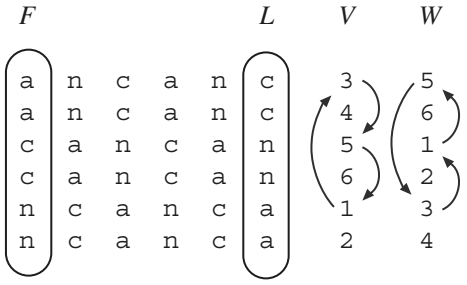

Fig. 2.9. The V and W arrays that are constructed for the string `cancan`

An even simpler case occurs when the coded text is just one character repeated many times; the decoder will only use the first occurrence of the character for all decoding.

It is worth being aware of this special case because it can also affect pattern matching. Of course, such a case is extremely unlikely to occur in practice. It might happen that a file containing just one repeated value is coded, such as the pixels in a blank image, but even in this case, just one different piece of information in the file, such as the image resolution, will prevent the rotations from being identical. If a particular algorithm is dependent on this not hap-

pening, it can be prevented by simply inserting one unique character (such as an end of string sentinel symbol) in T before it is transformed.

2.4 Further reading

The original Burrows and Wheeler paper (Burrows and Wheeler, 1994) remains an excellent explanation of the transform, and includes techniques for implementing the transform efficiently, including using a suffix tree in the forward transform, ways to use quicksort efficiently for the forward transform, and using counting rather than sorting in the reverse transform. Fenwick has published a series of papers which look in detail at implementation of the BWT; four early papers were mentioned in Chapter 1 (Fenwick, 1995b,c, 1996a,b); a summative paper can be found in the November 2007 special issue of *Theoretical Computer Science* about the BWT (Fenwick, 2007), which includes an algorithm for decoding a BWT file in natural order. The proposal for using run-length encoding to avoid the sorting problem in the BWT code was made in (Fenwick, 1996a). Fenwick's report also describes a private communication from Wheeler that gives an effective (if somewhat *ad hoc*) adaptation of quicksort that takes advantage of the particular structures available in the BWT.

The BWT is not the only way to permute texts and still be able to recover them, although other approaches are closely related. A number of such variants are described in Chapter 6.

The names of the arrays used in this chapter, and the rest of the book, differ slightly from some of those used in the BWT literature. This is explained in Appendix A; the main problem is that there is a conflict between the notation used in the pattern matching literature, and that used in the BWT literature. The use of F and L is consistent with Burrows and Wheeler's original paper; however, their T array corresponds to V in this chapter, since we use T for the input text.