THE BURROWS-WHEELER TRANSFORM: Data Compression, Suffix Arrays, and Pattern Matching

THE BURROWS-WHEELER TRANSFORM: Data Compression, Suffix Arrays, and Pattern Matching

Donald Adjeroh
West Virginia University

Tim Bell
University of Canterbury

Amar Mukherjee
University of Central Florida

Springer

Donald Adjeroh
West Virginia University
Morgantown, WV 26506
USA
adjeroh@csee.wvu.edu

Amar Mukherjee
University of Central Florida
Orlando, FL 32816-2362
USA
amar@eecs.ucf.edu

Tim Bell
University of Canterbury
Christchurch
New Zealand
tim.bell@canterbury.ac.nz

The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching
Donald Adjeroh, Tim Bell, and Amar Mukherjee

ISBN-13: 978-0-387-78908-8 e-ISBN-13: 978-0-387-78909-5

Library of Congress Control Number: 2008923556

Printed on acid-free paper.

9 8 7 6 5 4 3 2 1

springer.com

Preface

The Burrows-Wheeler Transform is one of the best lossless compression methods available. It is an intriguing — even puzzling — approach to squeezing redundancy out of data, it has an interesting history, and it has applications well beyond its original purpose as a compression method. It is a relatively late addition to the compression canon, and hence our motivation to write this book, looking at the method in detail, bringing together the threads that led to its discovery and development, and speculating on what future ideas might grow out of it.

The book is aimed at a wide audience, ranging from those interested in learning a little more than the short descriptions of the BWT given in standard texts, through to those whose research is building on what we know about compression and pattern matching. The first few chapters are a careful description suitable for readers with an elementary computer science background (and these chapters have been used in undergraduate courses), but later chapters collect a wide range of detailed developments, some of which are built on advanced concepts from a range of computer science topics (for example, some of the advanced material has been used in a graduate computer science course in string algorithms). Some of the later explanations require some mathematical sophistication, but most should be accessible to those with a broad background in computer science.

We have aimed to provide a detailed introduction to the current state of knowledge about the Burrows-Wheeler Transform. This ranges from explanations and examples of how the transform works, through analyzing the theoretical performance of the transform from various view points, to considering issues relevant to implementing it on "real" systems. Each chapter (except the last one) contains a "further reading" section to guide the reader around the large collection of literature that has explored the BWT in detail, and Appendix B points to ongoing research.

An important theme in this book is pattern matching and text indexing using the BWT. Because the transformed text contains a sorted version of the original text, it has considerable potential to help with locating patterns, and we look in detail at a number of variations that have been proposed and evaluated.

The BWT literature uses a variety of notation for the various structures used in the transform. Where possible we have tried to use standard notation, but unfortunately some key notations conflict with those used in the standard pattern matching literature, and so we have chosen to coin some new notations to avoid having the same notation with two meanings, at times in the same paragraph! Appendix A gives a summary of the notation used to avoid any confusion.

The BWT continues to be actively researched, and this book is merely a milestone in its history. Appendix B gives links to web sites that will be worth watching for future developments of the BWT and related systems.

We are also aware that despite some excellent help with checking this book, it will contain errors and require updates. An errata site is available at `http://www.cosc.canterbury.ac.nz/tim.bell/bwt/`. We welcome feedback on the book, and this can be sent to the authors via the contact details on this web site.

Acknowledgements

Many people have contributed to this book either directly or indirectly.

We have to first acknowledge the late David Wheeler, who conceived the idea on which this entire book is based. In researching the background of the BWT it has been inspiring to discover the role that this modest individual has played in making and influencing the history of computing in many areas, not just in data compression. Michael Burrows played an important role in developing and publishing the transform, and we have been very fortunate to receive valuable input from him while writing this book, including his insight into implementing the BWT on current and upcoming computer technology. We have also appreciated discussions with Dr. Joyce Wheeler, David Wheeler's wife, who has been able to help us with details of the history relating to the development of the transform. The photo of David Wheeler in Figure 1.3 was taken by Chris Hadley (University of Cambridge Computer Laboratory), and kindly supplied by Joyce Wheeler.

We also particularly wish to thank Peter Fenwick, who has been heavily involved in this book from the early stages of planning through to the final stages of checking. He has been a fount of wisdom, insight and information stemming from his long history of work on the Burrows-Wheeler Transform.

Our students who have worked with us on the BWT have our sincere thanks for a lot of detailed work and many useful discussions. We are particularly grateful to Andrew Firth, who performed an extensive comparison of BWT-based searching methods. Much of Chapter 7 draws on this work, and we are grateful for his permission to use it. Jie Lin, Fei Nan, Matt Powell, Ravi Vijaya Satya, Tao Tao, Nan Zhang and Yong Zhang have also contributed a

number of ideas that appear in this text. A preliminary version of some chapters in this book have been used in a graduate course on string algorithms, and we are grateful to the students for their comments and suggestions.

We have been fortunate to have many members of the BWT research community assist with advice and helping check parts of the book, particularly Paolo Ferragina, Craig Nevill-Manning, Giovanni Manzini, Alistair Moffat, Bill Smyth, Rossano Venturini, and Ian Witten. We also thank Ziya Arnavut, Mitsuharu Arimura and Kunihiko Sadakane, for providing us requested copies of their papers.

Many other people have contributed to practical aspects of this work: Jay Holland has lent us his sharp eye to assist with proof-reading — we appreciate his careful checking, and any errors are likely to have been introduced after he checked the text! Isaac Freeman has worked extensively on drawing and editing the figures for us; Julie Faris has helped with administrative aspects, Denise Tjon Ket Tjong with computer system support and help with technical issues, and Stacey Mickelbart has provided technical writing assistance. Amy Brais, our editor at Springer, has been most supportive in guiding us through the process of putting the book together.

Our respective universities, West Virginia University, the University of Canterbury, and the University of Central Florida have been very supportive of this project. Some of this work has been done while traveling, and we particularly acknowledge the Huazhong University of Science and Technology in Wuhan, China, which provided an excellent environment for writing.

The support from National Science Foundation grants (9977336, 0207819, 0312724, 0228370, 0312484) on data compression and compressed-domain pattern matching helped develop our interest in the Burrows-Wheeler Transform. A DOE CAREER grant (DE-FG02-02ER25541) provided support for studying the use of the BWT in biological sequence analysis. The NSF grant on "U.S., New Zealand and Australia Collaboration on Research on Data Compression" (0331188, 0331896) brought the authors of this book together in New Zealand to conceive and brainstorm this book.

The people who have made the greatest contribution, of course, are our families, who have released us for many long hours to write, re-write and fine tune this book. We are grateful to our wives, Leonie, Judith and Pampa, and other family members (some of whom have supported us for decades, and others who are only just about to learn what it means to have an academic for a parent), for their love and moral support throughout the project: Donald-Patrick (who was born during the writing of the book), Elise-Cindel, Andrew, Michael, Paula, Mita, Cecilia, Don and Nuella.

Morgantown, West Virginia *Don Adjeroh*
Christchurch, New Zealand *Tim Bell*
Orlando, Florida *Amar Mukherjee*

Contents

1 Introduction

The greatest masterpiece in literature is only a dictionary out of order.
Jean Cocteau

Here is a two word phrase in which the characters have been rearranged: `atd nrsoocimpsea`. Can you work out what the two words are that contain all these characters (including the space)? They could be `comedian pastors`, but they aren't. Nor are they `darpa economists`, `massacred potion`, `maniac doorsteps` or even `scooped martians`.

This puzzle is an example of the Burrows-Wheeler Transform (BWT), which uses the intriguing idea of muddling (we prefer to call it permuting) the letters in a document to make it easier to find a compact representation and to perform other kinds of processing. What is amazing about the BWT is that although there are 2,615,348,735,999 different ways to unmuddle the above characters into possible anagrams, the Burrows-Wheeler Transform makes it very easy to find the unique correct permutation very quickly.

The main point of permuting a text using the BWT is not to make it difficult to read, but to make it easy to compress. For example, for the following line from Hamlet's famous soliloquy:

> "To be or not to be: that is the question, whether tis nobler in the mind to suffer the slings and arrows of outrageous fortune."

we get the transformed text:

> "sdoosrtesrsefeeoe:nsrrtdn,r h onnhbhhbglfhuhnofu antttttw mltt bs ioaiui Tttn i fne r eoeetraoguiwi e ao es e. urqstoo o"

Notice that many characters in the transformed text appear in runs, or very close to previous occurrences. For longer texts this is even more noticeable; here is a typical excerpt from a Burrows-Wheeler Transform of all of Shakespeare's Hamlet:

```
nnnnnnnnnnnnnnnnnntnnnnnnnhnnngnnnnnnnnjnnnnnhdnnng
nnnnonnNnnnhhNnnnnnnnnntnnhnnnnnnnnnnnnnnNnndnnnhnn
nnnNnnnnnnnnnnnnnnnnnnnnnonntnnNNnnnnnnnndngnnnnnnn
nnnnnnnNnnnnnnnngnnnnnnnnnnnnnnnnnngnnnnnnnnonnnnnn
nnnNNnlnnnhnnnnnnnnnntdbdnnrrmnnmnmnnnuoccppppppdnr
rDolBbbdddodbbBddbbddbdBdbbdbdDddddBbbbbdDbubbdbdbB
```

This clustering of characters makes compression very easy. One simplistic way to code it would be to replace repeated characters with a number that says how many times it is repeated; for example, the first line above could be coded as:

```
19nt7nh3ng8nj5nhd3ng
```

In practice BWT coders use more sophisticated representations that take advantage of the mixture of frequently occurring characters (for example, the first four lines in the above example contain only 8 different characters, almost all of which are "n", "N", "h" or "g"). The point is that the transform makes the encoding task a lot simpler, and importantly, can give compression that is comparable with the very best lossless compression methods. Furthermore, it is generally faster than methods that give a similar amount of compression.

It has transpired that the BWT is useful for a lot more than compression because it contains an implicit sorted index of the input string. In this book we will review many of its other uses, especially for pattern-matching and full-text indexing, which leads to applications ranging from bioinformatics to machine translation.

The Burrows-Wheeler Transform method is often referred to as "block sorting", because it takes a *block* of text and permutes it. The main disadvantage of the block-wise approach is that it cannot process text character by character; it must read in a block (typically tens of kilobytes) and then compress it. This is not a limitation for most purposes, but it does rule out some applications that need to process data on-the-fly as it arrives. Another important point is that the text can be *sorted*; throughout this book we assume a unique ordering on the characters or symbols that are in the text so that substrings can be compared by the sorting algorithms. Most implementations work with a character set such as ASCII or 8-bit bytes, for which comparisons are trivial, but we shall see later that variations are possible where we take a more sophisticated approach to the ordering.

1.1 An example of a Burrows-Wheeler Transform

In this section we will give a simple example of how a text is transformed and reconstructed using the BWT. The method described here is for clarity of explanation, and in later chapters we will look at equivalent approaches that are a lot faster and simpler to implement, so don't be put off if it seems to be resource-hungry.

We will use a rather short block of text in this example: "aardvark$". The dollar sign is a sentinel, or end of string character, that we've added to simplify the explanation.

To generate the BWT, we list all nine rotations of the nine-character string, as shown in Figure 1.1a; that is, for every position in the string, we create a string of nine characters, wrapping around to the beginning if it runs off the end. The list is then sorted into lexical (dictionary) order (Figure 1.1b) (in this case we've assumed that `$` comes at the start of the lexical ordering). The transform is now complete, and the last column (i.e. last character of each row from top to bottom) is the output (Figure 1.1c).

```
aardvark$
ardvark$a
rdvark$aa
dvark$aar
vark$aard
ark$aardv
rk$aardva
k$aardvar
$aardvark
```

(a)

```
$aardvark
aardvark$
ardvark$a
ark$aardv
dvark$aar
k$aardvar
rdvark$aa
rk$aardva
vark$aard
```

(b)

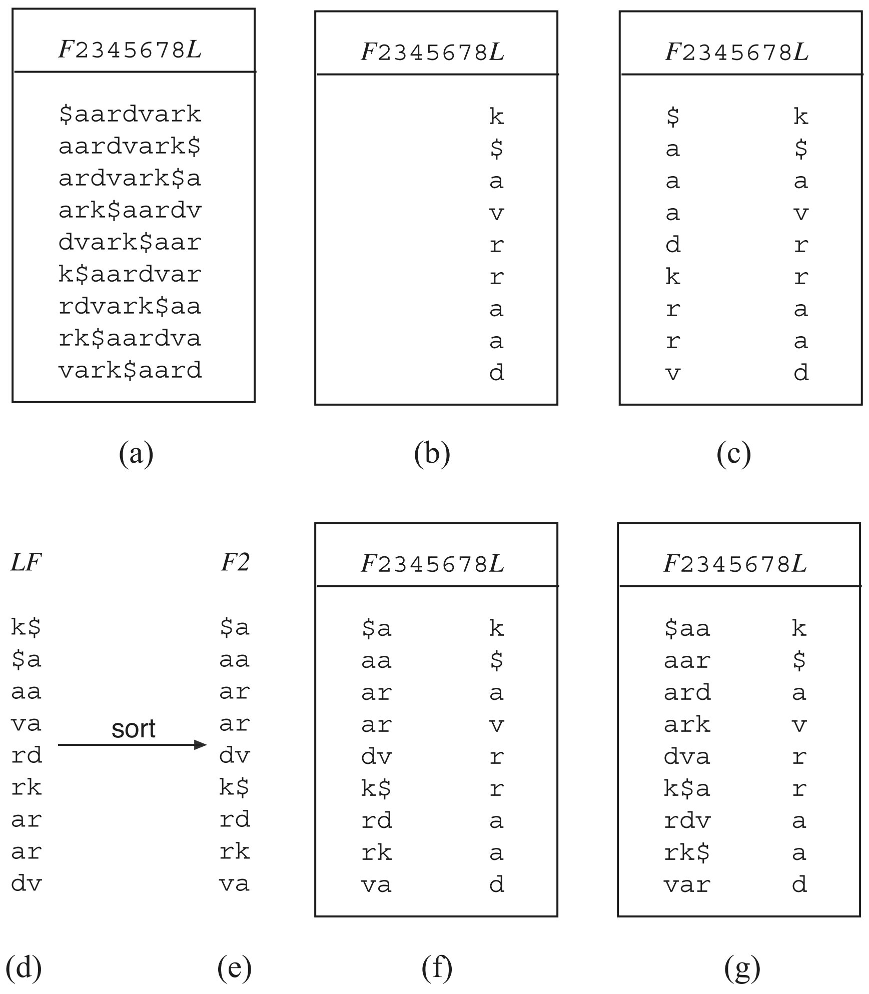
```
k$avrraad
```

(c)

Fig. 1.1. Burrows-Wheeler Transform of the string "aardvark$": (a) all rotations of the text are listed; (b) the list is sorted; (c) the last column is extracted as the BWT

The transform is that simple; in fact, in practice it is even simpler, as the substrings are never created, but are simply stored as references to positions in the original string. The size of the transformed text is identical to the original, and contains exactly the same characters but in a different order. This might seem to have achieved nothing, but as we shall see, it makes the text much easier to compress because it has drawn together characters that occur in related contexts — that is, characters that precede the same substrings.

It might seem that decoding the transformed text would be very difficult; after all, how do you "unmuddle" a list when there is an exponential number of ways to do it? The amazing thing about the BWT is that the reverse transform not only exists, but it can be done efficiently. A key observation is that we can reconstruct the list in Figure 1.1b, one column at a time. Figure 1.2a reproduces the list that we wish to construct, with the columns labeled. Traditionally we use F and L to label the first and last columns respectively; the others have been numbered for reference.



```
(a)            (b)            (c)
F2345678L      F2345678L      F2345678L
$aardvark              k      $       k
aardvark$              $      a       $
ardvark$a              a      a       a
ark$aardv              v      a       v
dvark$aar              r      d       r
k$aardvar              r      k       r
rdvark$aa              a      r       a
rk$aardva              a      r       a
vark$aard              d      v       d

(d)         (e)     (f)            (g)
LF          F2      F2345678L      F2345678L
k$          $a      $a      k      $aa     k
$a          aa      aa      $      aar     $
aa          ar      ar      a      ard     a
va   sort   ar      ar      v      ark     v
rd  ----->  dv      dv      r      dva     r
rk          k$      k$      r      k$a     r
ar          rd      rd      a      rdv     a
ar          rk      rk      a      rk$     a
dv          va      va      d      var     d
```

Fig. 1.2. Decoding the BWT: (a) the encoding information that we are trying to reconstruct; (b) the transformed BWT text in column L; (c) adding column F; (d) using L and F to extract all pairs of characters; (e) sorting the pairs; (f) adding the sorted pairs to the reconstruction; (g) adding sorted triples to the reconstruction

Column L is what the encoder sent to the decoder, so the reconstruction can start by filling column L (Figure 1.2b). Now observe that column F is simply all of the characters in the text in lexical order. Since the transformed text contains all of the characters, we can reproduce column F simply by sorting column L (Figure 1.2c).

Our next observation is that because of the wrap-around from the rotations used to generate the substrings, for a particular row, the character in column L must be followed by the one in column F in the original string. Thus we can find all pairs of characters in the original string by taking pairs from the last and then first columns (Figure 1.2d). If we sort these pairs (Figure 1.2e), they will give us the pairs in column 1 (F) and 2, and we now know three of the columns (Figure 1.2f).

Applying the wrap-around principle again, we can find all triples in the original text, sort them, and add them to the list (Figure 1.2g). We continue doing this until the whole list has been reproduced, giving us the information that the encoder had (Figure 1.2a). At this point it is trivial to read off the original string; we can take any row, and starting after the end-of-file symbol, read the characters, wrapping around at the end of the row.

This may seem like a lot of work to do the decoding. In practice most of the process just described is unnecessary and decoding can be done in $O(n)$ time by creating an auxiliary array that enables us to navigate around the transformed text. This is covered in detail in Chapter 2, but in the meantime, we will observe that the relationships just described mean that we can easily match the characters in columns L and F.

The transform that we have just described doesn't change the size of the file that has been transformed. However, when it is done to large files, we shall see that it makes the file a lot easier to compress because we end up with a very obvious clustering of characters.

1.2 Genesis of the Burrows-Wheeler Transform

The Burrows-Wheeler Transform is one of the most effective text compression methods to come out of the 20th century, yet its intriguing method of compression and its unusual history have meant that it was almost overlooked!

Data compression has turned out to be fundamental to getting things done on digital devices. Without MP3 files we couldn't download music or carry lots of songs in portable devices; without JPEG files digital cameras would only take a few shots before filling up and photos on web pages would take forever to load; and without the MPEG standard DVDs would only hold a few minutes of movies and the phrase "viral video" would never have been coined.

In this book we focus on *lossless* methods, which are able to decompress a file to exactly the same as it was before being compressed. However, many *lossy* methods (which are typically used for sound and images) rely on lossless methods in their final stage.

Compression on computers spans the second half of the 20th century. Shannon's ground-breaking paper on information theory is generally regarded as the foundation of compression systems (Shannon, 1948). The paper included a proposed coding method that has come to be known as Shannon-Fano coding, which was one of the earliest methods used to take advantage of some characters being more likely than others. Shannon-Fano coding is suboptimal, and it was one of Fano's students, David Huffman, who in 1952 published his well-known algorithm (Huffman, 1952), which became a stock technique and is still used today as a part of many kinds of compression system, including general-purpose lossless methods and systems for compressing audio and images. The next major improvements in compression performance came in the late 1970s, when Ziv and Lempel published the "LZ" methods which are still widely used in formats such as GIF and PNG images, as well as the ZIP and GZIP utilities (Ziv and Lempel, 1977, 1978). The LZ family of methods became popular because it gave excellent compression and yet was practical to run on computers at the time. By the time the 1980s arrived, Rissanen and Langdon (1979) had published a significant improvement on Huffman coding, called "Arithmetic Coding"[1]. This opened up a new way of looking at compression, and became the basis of a new wave of compression methods in the mid 1980s that used sophisticated models of text to achieve a new level of compression by "predicting" what the next character would be. At the time these methods were too resource intensive to be used as a utility, but they provided a new benchmark for compression performance. Of particular note was the PPM method, developed by Cleary and Witten (1984), and several subsequent variations that set new records for the amount of compression that could be achieved.

Arguably the last 20th century breakthrough in general purpose lossless compression methods was Burrows and Wheeler's enigmatic transform, the BWT. David Wheeler had come up with the transform as early as 1978, but it wasn't until 1994 that, with the help of Mike Burrows, the idea was turned into a practical data compression method, which was then published in a Digital Systems Research Center (Palo Alto) research report (Burrows and Wheeler, 1994). Their "block-sorting code", also dubbed the "Burrows-Wheeler Transform", left compression practitioners scratching their heads, as it involved rearranging the characters in a text before encoding, and then magically arranging them back in their original order in the decoder. The fact that the original can be re-created at all is somewhat astonishing, and their early work took some time to receive the recognition it deserved. Within a couple of years several authors and programmers had picked up the idea, apparently mainly through publications by Peter Fenwick (Fenwick, 1995b,c,

[1] Peter Elias had come up with the idea some time earlier, but apart from a brief mention in Abrahamson's 1963 book *Information Theory and Coding*, it did not get published as a feasible coding method until Rissanen and Langdon's paper appeared.

1996a,b) which led to Julian Seward's BZIP implementation. Around the same time there was a writeup by Mark Nelson in *Dr Dobb's Journal* (Nelson, 1996), and the BWT also appeared through informal channels such as on-line discussion groups.

Burrows and Wheeler have other significant achievements in the field of computing. David Wheeler (1927–2004) had a distinguished career, having worked on several early computers, including EDSAC which, in 1949, became the first stored program computer to be completed. Wheeler invented a method of calling closed subroutines which led to having a library of carefully tested subroutines, a concept that has been crucial for breaking down complexity in computer programming. Together with Maurice Wilkes and Stanley Gill, in 1951 he published the first book on digital computer programming[2]. He also did important work in cryptography, including the "Tiny Encryption Algorithm" (TEA), an encryption system that could be written in just eight lines of code, which made a mockery of US regulations that controlled the export of encryption algorithms — this one was small enough to memorize! Wheeler also designed and commissioned the first version of the Cambridge Ring, an experimental local network system based on a ring topology.

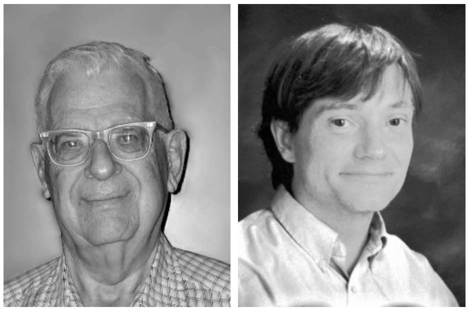

(a) (b)

Fig. 1.3. (a) David Wheeler (b) Michael Burrows

His work on compression developed during his time as a research consultant at Bell Labs (Murray Hill, N.J.) in 1978 and 1983. He retired in 1994 (the same year that the seminal BWT paper was released). His distinctions include being a Fellow of the Royal Society (1981), and a Fellow of the ACM (1994).

Michael Burrows also has a high profile outside his contribution to the BWT. He is one of the main people who developed the AltaVista search

[2] *The Preparation of Programs for an Electronic Digital Computer*, published by Addison-Wesley Press, Cambridge.

engine in 1995, which represented the state of the art prior to the arrival of Google's search engine. He later worked for Microsoft, and in 2007 is a senior researcher working at Google on their distributed infrastructure. Burrows had been supervised by Wheeler in the mid-1980s doing a PhD at Cambridge, and then went to work at Digital in the US. Wheeler had invented the transform in the 1970s, but it wasn't until he visited Digital in Palo Alto and then worked remotely with Burrows by email in 1990 that it was finally written up as a compression method.

In the late 1990s BWT was still regarded as being too slow for many applications, but its compression performance became well understood. Wheeler's "bred" (block reduce) and "bexp" (block expand) programs provided a publicly available implementation of the BWT method that proved the concept, but it was Julian Seward's efficient implementation as a general purpose utility called BZIP in 1996 that established BWT as something that had practical utility. A new version of Seward's utility called BZIP2 is now widely used because on today's hardware it can compress large files at speeds that are quite acceptable for interaction, to a smaller size than other widely used general purpose methods. For example, the 4 Mbyte file "bible.txt" from the Canterbury corpus can be compressed by BZIP2 in about 2 seconds on a 2.4 GHz computer, and decompressed in about 1 second. The GZIP utility compresses about three times as fast (and decompresses an order of magnitude faster), but the GZIP file is 40% larger than the BZIP2 one. Interestingly, BZIP2 combines one of the most recent compression breakthroughs (BWT) with one of the first (Huffman coding).

By the late 1990s researchers began to realize that the BWT approach might be useful for more than just compressing text. Because the BWT happens to "sort" the text into alphabetical order, the permuted text has the added benefit of acting as a kind of dictionary for the original text. Traditionally an index and the compressed text would be stored separately, even though they contain effectively the same information. In this light, the BWT is an intermediate representation that is halfway between a text and an index; the original text can be reconstructed efficiently from it, yet sorted lists like the one shown in Figure 1.1b are ripe for binary searching, giving very fast searching for arbitrary fragments in the text.

In this book we explore this intriguing view of a transformed file as both the text and an index, and look at applications that exploit this. But first let's take a look at some key ideas behind the BWT: transformation, permutation, and recency.

1.3 Transformation

Suppose you had to calculate, in Roman numerals, the sum MCMXCIX + I. Perhaps you know a method for adding Roman numerals, but chances are that you would have transformed the problem into a more familiar notation: 1999

+ 1. The sum is now easily calculated, and the answer in Roman numerals is obtained by a reverse transform, as shown in Figure 1.4.

$$\begin{tikzcd}
\text{MCMXCIX} + \text{I} \arrow[r, "\text{transform}"] \arrow[d] & 1999 + 1 \arrow[d, "\text{calculate in easier domain}"] \\
\text{MM} & 2000 \arrow[l, "\text{reverse transform}"']
\end{tikzcd}$$

Fig. 1.4. Calculating MCMXCIX + I using a transform

Different representations have different strengths; Roman numerals might not seem that easy to work with, but they look impressive, and some say that they are used to show the dates in movie and TV credits to make it difficult for a casual viewer to determine how old the film is.

Transformations have long been put to more practical uses in engineering, to convert a representation to a “space” in which it is easier to work with. One of the best known is the Fourier transform, which converts a signal into the sum of a set of sine waves. In this format, it is easy to perform operations such as boosting the bass in an audio signal (just increase the amplitude of the low frequency sine waves), or to find areas in an image with a lot of detail (look for high frequency sine waves with a high amplitude).

Transformations related to the Fourier transform, especially the Discrete Cosine Transform (DCT), have long been used in lossy compression methods for audio and image compression, such as MP3 and JPEG. Viewing a signal as a sum of cosine waves makes it easy to compress because it is possible to decrease the level of detail stored, especially for components that are difficult to hear or see — in fact, some frequencies could even be eliminated. The information is also easy to decompress, as it is simply the sum of the frequency components.

Transforms open up new ways to manipulate and store data, in the same way as the language one is using can affect the way that we understand our world (the Sapir–Whorf hypothesis). Or more bluntly, when the only tool that you have is a hammer, every problem looks like a nail. A transformation gives us a new tool to solve a problem, a new language to describe what we can do with the data.

Generally a transform doesn’t change the amount of data used to represent a signal; it just gives us a new way of looking at it. Here, any compression happens after the transformation, and is done either by exploiting patterns exposed by the transformation, or by using a less accurate representation for components in a way that is not likely to be perceived by a human.

The Burrows-Wheeler Transform was a breakthrough because it provided a reversible transformation for text that made it significantly easier to compress. There are many other reversible transformations that could be applied to a text — for example, the characters could be stored backwards, or the first two letters after each space could be transposed — but these don't help us to compress the text. The power of the BWT is that it pulls together related characters, in the same way that a Fourier transform separates out high-frequency components from low-frequency ones.

For example, Figure 1.5 shows a segment of a BWT-sorted file for Shakespeare's Hamlet. It is sorted into lexical order, starting at the first (F) column. Because each row of the table is generated by wrapping around the original text, the last (L) column is actually the character that comes *before* the one in the F column. So from the figure we can see that "ot " is normally preceded by n, but occasionally by h, g or j. It now becomes clear why we get so much repetition in the transformed file; the characters are clustered according to what words or phrases they are likely to precede — u is likely to precede estion, m or w are likely to precede ent, and so on. Some characters are very predictable — osencrantz and Guildenstern is always preceded by an R, while others are less so — est occurs in Hamlet preceded by every letter of the alphabet except a, o, q, v, x, y and z.

F	. . . L
ot look upon his like again.	. . . n
ot look upon me; Lest with th	. . . n
ot love on the wing,-- As I p	. . . h
ot love your father; But that	. . . n
ot made them well, they imita	. . . n
ot madness That I have utter'	. . . n
ot me'? Ros. To think, my lor	. . . n
ot me; no, nor woman neither,	. . . n
ot me? Ham. No, by the rood,	. . . g
ot mend his pace with beating	. . . n
ot mine own. Besides, to be d	. . . n
ot mine. Ham. No, nor mine no	. . . n
ot mock me, fellow-student. I	. . . n
ot monstrous that this player	. . . n
ot more like. Ham. But where	. . . n
ot more native to the heart,	. . . n
ot more ugly to the thing tha	. . . n
ot more, my lord. Ham. Is not	. . . j
ot move thus. Oph. You must s	. . . n
ot much approve me.--Well, si	. . . n

Fig. 1.5. Part of the BWT sorted list for Shakespeare's Hamlet

1.4 Permutation

Permutations are rearrangements of the order of symbols, such as the rearrangement of letters in anagrams which we have already mentioned (for example "eleven plus two" is an anagram of "twelve plus one"). Traditionally permutations don't allow the repetition of a symbol — in fact, a mathematical permutation is a subset of symbols taken from a set of distinct symbols. In the context of this book we are interested in rearrangements of a string that can contain duplicate characters.

If duplicates are not allowed then the number of permutations of n symbols is simply $n!$, the factorial of n. For example, the 6 characters `abcdef` can be arranged $6! = 720$ ways. Allowing duplicates reduces the number of permutations; in the extreme, a string such as `aaaaaa` which contains only one distinct character has only one permutation. In general, if we have n characters in the text, with one character occurring n_1 times, another n_2 times and so on, then the number of permutations possible is $\frac{n!}{n_1!n_2!...n_k!}$. Hence for our opening example, `atd nrsoocimpsea`, we have $n = 16$, three of the n_i values are 2 (for `a`, `s` and `o`), and the rest are 1, giving us $\frac{16!}{2.2.2}$ =2,615,348,736,000 possible permutations (including the unpermuted text itself). The number of permutations for a text will generally exhibit a combinatorial explosion of possibilities, which makes the existence of the reverse BWT all the more surprising.

Permutations have been a staple method for encryption, and are featured in the widely used "Advanced Encryption Standard" (AES), and its 1976 predecessor, the "Data Encryption Standard" (DES). In encryption, the function of permutation is to remove any clues that might be obtained by the juxtaposition of characters. It is somewhat ironic that the Burrows-Wheeler Transform, which also permutes the text, has almost the opposite purpose, as it highlights the regularities of adjacent characters. It may even be that one of the reasons that the BWT was initially viewed with some suspicion is that the main application of permutations in coding up to that time had been to make it impossible to reverse the coding. The connection with encryption is intriguing because Burrows also developed the "Tiny Encryption Algorithm" (TEA) mentioned earlier, which is based on a similar structure to DES and AES.

Two special cases of a permutation arise in the process of performing the Burrows-Wheeler Transform. One is the *circular shift* permutation, which can be seen in the rows of Figure 1.1a, where all of the characters are moved one position to the left, and the first character moves to the last position. A text of n characters usually has n circular shift permutations, although if the text is entirely composed of repeated substrings (such as `blahblahblah`) then some of the n circular shifts will produce the same string. This situation is very unlikely to occur in practice (the most likely case being a file containing only a single character repeated many times), but it is a case which causes unusual behavior for the BWT.

The other kind of permutation that arises in the BWT is one found in the *columns* of a sorted list such as the one in Figure 1.1b. Each column is also a permutation of the input text, with the first one containing all identical characters grouped together. This column is the result of sorting the input characters, and indeed sorting is a special case of permutation. The last column is the output of the transform, and is the one permutation of the text that we are the most interested in. The BWT uses this particular permutation which is dictated by the sort order, but later we will look at methods that use slightly different permutations based on different ways of comparing substrings of the text.

Finally, a trivial permutation which comes up when discussing the Burrows-Wheeler Transform is the reverse of the input string. The simplest implementation of the BWT will output the file in reverse order, although this is easily avoided by reversing the input when it is read into memory before encoding, or reversing the output from the decoder. In general reversing a string does not affect compression performance, but in some practical situations it can. This is discussed in Section 2.2.

1.5 Recency

In the physical world, it's often efficient to keep recently used documents, equipment or other resources nearby on the basis that the most recently used items are the most likely to be used again. Of course, one can argue the opposite: if something has been used a lot recently then perhaps we will be finished with it soon! In practice the recency effect is a safe observation to take advantage of, and the output of the Burrows-Wheeler Transform very much amplifies any recency effects in the text by bringing together characters that have occurred in related contexts.

The traditional use of the recency effect on computers is the LRU (least recently used) mechanism for caching: when data needs to be displaced from high-speed memory, we generally favor discarding the data that has been used furthest in the past. The extreme form of a recency mechanism is the stack, which allows access to only the *most* recently used item. While this might seem limiting, the stack is a very powerful construct, especially for the complex task of parsing recursively structured input such as programming languages; and of course, the stack is fundamental to most programming language implementations for allowing recursive function invocations.

There are various ways to take advantage of the recency effect of the BWT output, and these are discussed in detail in Chapter 3. The original BWT paper used a "move-to-front" (MTF) system where the shortest codes are allocated to the characters at the "front" of a list. When a character is to be coded, its position in the list is transmitted and then it is moved to the front of the list, thereby demoting all the other characters that were ahead of it in the list. Variations of this approach have been used very successfully with the BWT.

To implement the MTF system, the compression of the BWT output could be done by simply storing how many different characters have been encountered since the previous occurrence of the current character. For example, if the text `abbc` has just been decoded then if a 2 is received next it would represent an `a` (because you would need to skip two *different* characters to get to the previous `a`), while a `b` would be coded as a 1, and `c` as a 0. Very small numbers will be common in the output from the MTF system, and these numbers are then represented by codes that use fewer bits for smaller numbers, and more bits for the larger ones.

An alternative approach which has found favor in recent years for compressing the BWT output avoids using the move-to-front strategy to capture the recency effect; we simply use a conventional coder (adaptive Huffman or arithmetic coding) and bias the probabilities to favor recent occurrences of characters. Since the coders work with estimated probabilities, we just need a system that estimates high probabilities for characters that have occurred a lot recently, since the coder will use shorter codes for the high probability events. This is done by having recent occurrences of a character contribute significantly more to its estimated probability than past ones by reducing the weight of "old" characters. For the BWT this bias for recency has to be very strong, as repeated characters can occur in relatively small clusters. This will be discussed in more detail in Section 3.2.

1.6 Pattern matching

Compression and pattern matching are closely related. One way of looking at a compression method is that it simply looks for patterns, and takes advantage of them to remove repetition. For example, Ziv-Lempel methods search previous sections of a text for matches; if Shakespeare's "Hamlet" is being compressed[3], and the next string to be encoded is the 18th occurrence of the string "noble", the system will search to find that the string occurred 1366 characters earlier, and can replace it with a reference that points back 1366 characters, and gives the length of the match (5 characters). In other compression methods the pattern is a *context* that is being searched for, to make predictions based on what has happened in past occurrences of the context — for example, a compression system might want to know what character is most likely to come after "noble", and could find this out by locating all previous occurrences of "noble" which will reveal that 16 of the 17 previous occurrences were followed by a space, and one was followed by an "r".

Because the compression process involves pattern matching, it makes sense to try to harness all the searching done during compression if a user wants

[3] There are several versions of Hamlet available; these statistics are for a particular version from Project Gutenberg.

to search for a key in the compressed text. This means that we might be able to search a compressed document without decompressing it, which is "compressed-domain searching". Simplistically, one might compress the search key, and try to find the encoded key in the compressed file. Unfortunately this is unlikely to work in practice because the encoding of a substring can depend on other text surrounding it, although a number of algorithms have been developed for compression methods that are able to work around this.

For the Burrows-Wheeler Transform, however, the matching process is much simpler, at least in principle, because the encoding is based on sorting every substring of the text into lexical order — we have a sorted list (ideal for binary search) available as a by-product of compression! For example, Figure 1.6 shows some of the sorted strings that are generated during the BWT encoding of Shakespeare's Hamlet. Of course, the full substrings aren't actually generated; they are simply a list of references to positions in the original text. The L column (which shows the BWT output[4]) is really just the character in the original string that comes before the one in the F column. What makes searching in the Burrows-Wheeler Transformed text easy is that using an auxiliary array that is needed for decoding, the rows in the list can be accessed randomly, and characters in each row are easily read off in linear time. Thus, without fully decoding the text we are able to perform a binary search of the original text.

For example, if we were to search for the word "nobler" in the text, we would begin by decoding the middle few characters of the sorted list ("there's a special providence...") and discover that "nobler" is lexically earlier in the file. Carrying on with the binary search brings us to the section in Figure 1.6, and consequently to the line beginning "nobler in the mind to suffer...", which can be decoded for as many characters as are required to show the matched part of the text.

From this point of view, the compressed text is like a wound-up spring, containing lexical energy added by the sorting during encoding, and waiting to be released in a search.

1.7 Organization of this book

Now that we have looked informally at how the BWT can achieve compression, yet still allow efficient searching, in the next chapter we will describe in some detail how the BWT is implemented in practice, including data structures for doing the transformation quickly, and for reversing it efficiently. Chapter 3 will consider what to do with the transformed text, as there are a variety of methods that can be used to code the very repetitive text that is generated.

Chapter 4 looks at suffix trees and suffix arrays, which are important ideas in compression and pattern matching. They pre-date the Burrows-Wheeler

[4] The Hamlet text is similar in length to the block size used by BWT coders, so the L column shows the level of repetition typical of the output of a BWT coder.

F	...	*L*
no_sooner_shall_the_mountains	...	_
no_spirit_dare_stir_abroad;_T	...	_
no_such_stuff_in_my_thoughts.	...	_
no_such_thing?_Laer._Know_you	...	_
no_tokens._Which_done,_she_to	...	_
no_tongue,_Nor_any_unproporti	...	_
no_tongue,_will_speak_With_mo	...	_
no_tongue:_I_will_requite_you	...	_
no_tongues_else_for's_turn._H	...	_
no_touch_of_it,_my_lord._Ham.	...	_
no_truant._But_what_is_your_a	...	_
no_wind_shall_breathe;_But_ev	...	_
no_words_of_this;_but_when_th	...	_
nobility_of_love_Than_that_wh	...	_
noble_Hamlet:_Mine_and_my_fat	...	_
noble_and_most_sovereign_reas	...	_
noble_dust_of_Alexander_till_	...	_
noble_father_in_the_dust:_Tho	...	_
noble_father_lost;_A_sister_d	...	_
noble_father_slain_Pursu'd_my	...	_
noble_father's_person,_I'll_s	...	_
noble_heart.--Good_night,_swe	...	_
noble_in_reason!_how_infinite	...	_
noble_lord?_Hor._What_news,_m	...	_
noble_mind_is_here_o'erthrown	...	_
noble_mind_Rich_gifts_wax_poo	...	_
noble_rite_nor_formal_ostenta	...	_
noble_son_is_mad:_Mad_call_I_	...	_
noble_substance_often_doubt_T	...	_
noble_youth,_The_serpent_that	...	_
noble_youth:_mark._Laer._What	...	_
nobler_in_the_mind_to_suffer_	...	_
noblest_to_the_audience._For_	...	_
nocent_love,_And_sets_a_blist	...	n
nock_him_about_the_sconce_wit	...	k
nocked_about_the_mazard_with_	...	k
nocking_each_other;_And_with_	...	k
noculate_our_old_stock_but_we	...	i
nod,_take_away_her_power;_Bre	...	y
nods,_and_gestures_yield_them	...	_
noint_my_sword._I_bought_an_u	...	a

Fig. 1.6. Another part of the BWT sorted list for Shakespeare's Hamlet; spaces are shown as an underscore

Transform, which is very similar to a suffix array, and it is valuable to study them to help understand the BWT better.

Chapter 5 reviews theoretical results for BWT-based schemes, such as universal compression, optimality issues, and computational complexity. It also covers current challenges in improving the BWT algorithm, with respect to compression performance, theoretical space and time complexity. This chapter also explores the connection between the BWT and other compression algorithms, such as PPM (Prediction by Partial Matching), DMC (Dynamic Markov Compression) and LZ (Ziv-Lempel) coding.

Chapter 6 will discuss other approaches that are very closely related to the BWT. This will include members of the class of compression algorithms that perform compression based on sorted contexts, such as permutation-based coding, block-sorting schemes, and newer approaches such as word-based BWT.

Chapter 7 introduces the problem of pattern matching, and some standard algorithms for searching uncompressed text. We then look at methods that perform searching with the aid of the BWT, including both methods that store indexes as part of the BWT-based compression scheme, and those that perform searching with limited partial decompression of the BWT. These methods exploit the sorted contexts used by BWT and other members of this class of compression algorithm. The remainder of the chapter moves away from exact matching, and presents several algorithms for approximate pattern matching, longest common subsequence and sequence alignment, including algorithms for approximate pattern matching using the BWT. It also briefly considers hardware-based methods for pattern matching.

Chapter 8 explores emerging applications of the BWT, different from text compression and text pattern matching, such as using the BWT for compressed suffix arrays and compressed suffix trees, compressed full-text indexing, image compression, shape analysis, DNA sequence analysis in bioinformatics, and entropy estimation.

We conclude in Chapter 9 with an overview of the BWT with speculation on the short- and long-term direction of research work on BWT.

1.8 Further reading

The "Further reading" section at the end of each chapter will provide key references and tangential information that may be relevant to those wanting to study the topic of the chapter further.

The key reference for this book is Burrows and Wheeler's original 1994 paper titled "A block-sorting lossless data compression algorithm" (Burrows and Wheeler, 1994). Early descriptions of the method were written by Fenwick, initially in three technical reports (Fenwick, 1995b,c, 1996a), and then in a 1996 article in the *Computer Journal* (Fenwick, 1996b). Fenwick's work lead to Julian Seaward's BZIP program, which evolved into BZIP2, a widely used general-purpose implementation based on the BWT. A 1996 article by Mark Nelson in the *Dr Dobb's Journal* (Nelson, 1996) also helped to make the idea public. Soon after that papers about the BWT appeared in the Data Compression Conference (held annually in Snowbird, Utah) and the method became more widely understood. A survey article about the Burrows-Wheeler compression can be found in Fenwick (2003a). A meeting to mark the tenth anniversary of the BWT was held by the DIMACS Center at Rutgers University in August 2004, and a special edition of *Theoretical Computer Science* in November 2007 (volume 387, issue 3) is focused on the BWT. The special edition includes a foreword by Michael Burrows, which gives some interesting background to how the method was developed. It also includes three papers that provide useful overviews and analysis of BWT: Fenwick (2007), Kaplan et al. (2007), and Giancarlo et al. (2007).

The move-to-front (MTF) method used in the original BWT paper is based on work by Bentley et al. (1986) which uses the MTF list for compression, although in this case it was based on coding words rather than characters, and thus the MTF list had to be able to deal with a large vocabulary.

The puzzle at the start of the chapter is an anagram of `data compression`, which can be decoded using the inverse Burrows-Wheeler Transform[5]. It also happens to decode to "don amar to spices". Purists might have preferred us to use the example "The Magic Words are Squeamish Ossifrage" (made famous by the 1977 RSA cipher challenge). Interestingly "Squeamish Ossifrage" has an anagram relevant to data compression: "I squish for a message". However, the BWT of "squeamish ossifrage" is "hreugiassma sfiseoq". The example used from Shakespeare ("To be or not to be...") also has an interesting anagram, discovered by Cory Calhoun: "In one of the Bard's best-thought-of tragedies, our insistent hero, Hamlet, queries on two fronts about how life turns rotten."

Shannon's original 1948 paper that is the basis of much of the work in data compression was published in the *Bell System Technical Journal* (Shannon, 1948), and subsequently in a book by Shannon and Weaver (1949). Other important milestones in data compression prior to the Burrows-Wheeler Transform were Huffman's codes (Huffman, 1952), Ziv and Lempel's methods (Ziv and Lempel, 1977, 1978), arithmetic coding (Pascoe, 1976; Rissanen, 1976; Rissanen and Langdon, 1979), and "Prediction by Partial Matching" (Cleary and Witten, 1984). General texts about data compression include Storer (1988), Bell et al. (1990), Nelson and Gailly (1995), Williams (1991), Witten et al. (1999), Sayood (2000), Moffat and Turpin (2002), Sayood (2003) and Salomon (2004).

[5] Actually, the transform gives only the order of the letters; some extra information is needed to establish which letter is the starting point, but it is a puzzle after all!

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 $R[1 \dots n]$ 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.

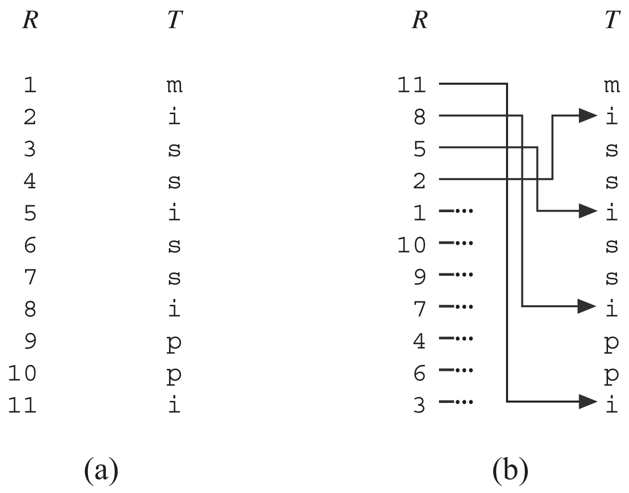


Fig. 2.2. The R array used to sort the sample file `mississippi`

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`