The Burrows-Wheeler Transform

Genome compression

Suffix arrays have greatly reduced the memory required for efficient text searches, and until the start of this century, they represented the state of the art in pattern matching. Can we be so ambitious as to look for a data structure that would encode *Text* using memory approximately equal to the length of *Text* while still enabling fast pattern matching?

To answer this question, we will digress to consider the seemingly unrelated topic of **text compression**. In one simple compression technique called **run-length encoding**, we replace a **run** of k consecutive occurrences of symbol s with only two symbols: k, followed by s. For example, run-length encoding would compress the string `TTTTTGGGAAAACCCCCCA` into `5T3G4A6C1A`.

Run-length encoding works well for strings having lots of long runs, but real genomes do not have many runs. What they do have, as we saw in Chapter 3, are *repeats*. It would therefore be nice if we could first manipulate the genome to convert repeats into runs and then apply run-length encoding to the resulting string.

A naive way of creating runs in a string is to reorder the string's symbols lexicographically. For example, `TACGTAACGATACGAT` would become `AAAAACCCGGGTTTT`, which we could then compress into `5A3C3G4T`. This method would represent a 3 GB human genome file using just four numbers.

STOP and Think: What is wrong with applying this compression method to genomes?

Ordering a string's symbols lexicographically is not suitable for compression because many different strings will get compressed into the *same* string. For example, the DNA strings `GCATCATGCAT` and `ACTGACTACTG` — as well as any string with the same nucleotide counts — get reordered into `AAACCCGGTTT`. As a result, we cannot **decompress** the compressed string, i.e., invert the compression operation to produce the original string.

Constructing the Burrows-Wheeler transform

Let's consider a different method of converting the repeats of a string into runs that was proposed by Michael Burrows and David Wheeler in 1994. First, form all possible **cyclic rotations** of *Text*; a cyclic rotation is defined by chopping off a suffix from the end of

Text and appending this suffix to the beginning of *Text*. Next — similarly to suffix arrays — order all the cyclic rotations of *Text* lexicographically to form a $|Text| \times |Text|$ matrix of symbols that we call the **Burrows-Wheeler matrix** and denote by M(*Text*) (Figure 9.8).

Cyclic Rotations	*M*("panamabananas$")
panamabananas$	$ p a n a m a b a n a n a **s**
$panamabananas	a b a n a n a s $ p a n a **m**
s$panamabanana	a m a b a n a n a s $ p a **n**
as$panamabanan	a n a m a b a n a n a s $ **p**
nas$panamabana	a n a n a s $ p a n a m a **b**
anas$panamaban	a n a s $ p a n a m a b a **n**
nanas$panamaba	a s $ p a n a m a b a n a **n**
ananas$panamab	b a n a n a s $ p a n a m **a**
bananas$panama	m a b a n a n a s $ p a n **a**
abananas$panam	n a m a b a n a n a s $ p **a**
mabananas$pana	n a n a s $ p a n a m a b **a**
amabananas$pan	n a s $ p a n a m a b a n **a**
namabananas$pa	p a n a m a b a n a n a s **$**
anamabananas$p	s $ p a n a m a b a n a n **a**

FIGURE 9.8 All cyclic rotations of "panamabananas$" (left) and the Burrows-Wheeler matrix M("panamabananas$") of all lexicographically ordered cyclic rotations (right). BWT("panamabananas$") is the last column of M("panamabananas$"): **"smnpbnnaaaaa$a"**.

Notice that the first column of M(*Text*) contains the symbols of *Text* ordered lexicographically, which is just the naive rearrangement of *Text* that we already described. In turn, the second column of M(*Text*) contains the second symbols of all cyclic rotations of *Text*, and so it too represents a (different) rearrangement of symbols from *Text*. The same reasoning applies to show that any column of M(*Text*) is some rearrangement of the symbols of *Text*. We are interested in the last column of M(*Text*), called the **Burrows-Wheeler transform** of *Text*, or BWT(*Text*), which is shown in red in Figure 9.8.

STOP and Think: We have seen that the first column of M(*Text*) cannot be uniquely decompressed to yield *Text*. Do you think that some other column of M(*Text*) can be inverted to yield *Text*?

Burrows-Wheeler Transform Construction Problem:
Construct the Burrows-Wheeler transform of a string.

Input: A string *Text*.
Output: BWT(*Text*).

STOP and Think: Figure 9.8 suggests a simple algorithm for computing BWT(*Text*) based on constructing M(*Text*). Can you construct BWT(*Text*) using less memory given *Text* and SUFFIXARRAY(*Text*)?

From repeats to runs

If we re-examine the Burrows-Wheeler transform in Figure 9.8, we immediately notice that it has created the run "aaaaa" in BWT("panamabananas$") = "smnpbnnaaaaa$a".

STOP and Think: Why do you think that the Burrows-Wheeler Transform produced this run?

Imagine that we take the Burrows-Wheeler transform of Watson and Crick's 1953 paper on the double helix structure of DNA. The word "and" is repeated often in English, which means that when we form all possible cyclic rotations of the Watson & Crick paper, we will witness a large number of rotations beginning with "and..." In turn, we will observe many rotations that begin with "nd..." and end with "...a". When all the cyclic rotations of *Text* are sorted lexicographically to form M(*Text*), all rows that begin with "nd..." and end with "...a" will tend to clump together. As illustrated in Figure 9.9, this clumping produces runs of "a" in the final column of M(*Text*), which we know is BWT(*Text*).

The substring "ana" in "panamabananas$" plays the role of "and" in Watson and Crick's paper and explains three of the five occurrences of "a" in the repeat "aaaaa" in BWT("panamabananas$") = "smnpbnnaaaaa$a". When the Burrows-Wheeler transform is applied to a genome, it converts the genome's many repeats into runs. As we already suggested, after applying the Burrows-Wheeler transform, we can apply an additional compression method such as run-length encoding in order to further reduce the memory.


```
nd Corey (1).  They kindly made their manuscript availa ...... a
nd criticism, especially on interatomic distances.  We  ...... a
nd cytosine.  .The sequence of bases on a single chain d ...... a
nd experimentally (3,4) that the ratio of the amounts o ...... u
nd for this reason we shall not comment on it.  We wish ...... a
nd guanine (purine) with cytosine (pyrimidine).  In oth ...... a
nd ideas of Dr.  M. H. F. Wilkins, Dr.  R. E. Franklin  ...... a
nd its water content is rather high.  At lower water co ...... a
nd pyrimidine bases.  The planes of the bases are perpe ...... a
nd stereochemical arguments.  It has not escaped our no ...... a
nd that only specific pairs of bases can bond together  ...... u
nd the atoms near it is close to Furberg's 'standard co ...... a
nd the bases on the inside, linked together by hydrogen ...... a
nd the bases on the outside.  In our opinion, this stru ...... a
nd the other a pyrimidine for bonding to occur.  The hy ...... a
nd the phosphates on the outside.  The configuration of ...... a
nd the ration of guanine to cytosine, are always very c ...... a
nd the same axis (see diagram).  We have made the usual ...... u
nd their co-workers at King's College, London.  One of  ...... a
```

FIGURE 9.9 A few consecutive rows selected from M(*Text*), where *Text* is Watson and Crick's 1953 paper on the double helix. Rows beginning with "nd..." often end with "...a" because of the common occurrence of the word "and" in English, which causes runs of "a" in BWT(*Text*).

EXERCISE BREAK: There is only one run of length at least 10 in the *E. coli* genome. How many runs of length at least 10 do you find after applying the Burrows-Wheeler transform to the *E. coli* genome?

Inverting the Burrows-Wheeler Transform

A first attempt at inverting the Burrows-Wheeler transform

Before we get ahead of ourselves, remember that compressing a genome does not count for much if we cannot decompress it. In particular, if there exist a pair of genomes that the Burrows-Wheeler transform compresses into the same string, then we will not be able to decompress this string. But it turns out that the Burrows-Wheeler transform is reversible!

STOP and Think: Can you find the (unique) string whose Burrows-Wheeler transform is "enwvpeoseu$llt"? It could be "newtloveslupe$", "elevenplustwo$", "unwellpesovet$", or something else entirely.

Consider the toy example BWT(*Text*) = "ard$rcaaaabb". First, recall that the first column of M(*Text*) is the lexicographic rearrangement of symbols in BWT(*Text*), i.e., "$aaaaabbcdrr". For convenience, we will use the terms *FirstColumn* and *LastColumn* (i.e., BWT(*Text*)) when referring to the first and last columns of M(*Text*), respectively.

We know that the first row of M(*Text*) is the cyclic rotation of *Text* beginning with "$", which occurs at the end of *Text*. Thus, if we determine the first row of M(*Text*), then we can move the "$" to the end of this row and reproduce *Text*. But how do we determine the remaining symbols in this first row, if all we know is *FirstColumn* and *LastColumn*?

```
$ ? ? ? ? ? ? ? ? ? ? a
a ? ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? ? d
a ? ? ? ? ? ? ? ? ? ? $
a ? ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? ? c
b ? ? ? ? ? ? ? ? ? ? a
b ? ? ? ? ? ? ? ? ? ? a
c ? ? ? ? ? ? ? ? ? ? a
d ? ? ? ? ? ? ? ? ? ? a
r ? ? ? ? ? ? ? ? ? ? b
r ? ? ? ? ? ? ? ? ? ? b
```

STOP and Think: Using the first and last columns of the Burrows-Wheeler matrix shown above, can you find the first symbol of *Text*?

Note that the first symbol in *Text* must follow "$" in *any* cyclic rotation of *Text*. Because "$" occurs as the fourth symbol of *LastColumn* = "ard$rcaaaabb", we know that if we walk one symbol to the right from the end of the fourth row of M(*Text*), then we will "wrap around" and arrive at the fourth symbol of *FirstColumn*, which is "a" in "$aaaaabbcdrr". Therefore, this "a" belongs in the first position of *Text*:

```
$ a ? ? ? ? ? ? ? ? ? a
a ? ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? ? d
a ? ? ? ? ? ? ? ? ? ? $
a ? ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? ? c
b ? ? ? ? ? ? ? ? ? ? a
b ? ? ? ? ? ? ? ? ? ? a
c ? ? ? ? ? ? ? ? ? ? a
d ? ? ? ? ? ? ? ? ? ? a
r ? ? ? ? ? ? ? ? ? ? b
r ? ? ? ? ? ? ? ? ? ? b
```


STOP and Think: Which symbol is hiding in the second position of *Text*?

Following the same logic of "wrapping around", the next symbol of *Text* should be the first symbol in a row of M(*Text*) that ends in "a". The only trouble is that five rows end in "a", and we don't know which of them is the correct one! If we guess that this "a" is the seventh symbol of "ard$rcaaaabb", then we obtain "b" in the second position of *Text* (Figure 9.10 (left)). On the other hand, if we guess that this "a" is the ninth symbol of "ard$rcaaaabb", then we obtain "c" in the second position of *Text* (Figure 9.10 (middle)). Finally, if we guess that this "a" is the tenth symbol of "ard$rcaaaabb", then we obtain "d" in the second position of *Text* (Figure 9.10 (right)).

```
$ a b ? ? ? ? ? ? ? ? a     $ a c ? ? ? ? ? ? ? ? a     $ a d ? ? ? ? ? ? ? ? a
a ? ? ? ? ? ? ? ? ? ? r     a ? ? ? ? ? ? ? ? ? ? r     a ? ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? ? d     a ? ? ? ? ? ? ? ? ? ? d     a ? ? ? ? ? ? ? ? ? ? d
a ? ? ? ? ? ? ? ? ? ? $     a ? ? ? ? ? ? ? ? ? ? $     a ? ? ? ? ? ? ? ? ? ? $
a ? ? ? ? ? ? ? ? ? ? r     a ? ? ? ? ? ? ? ? ? ? r     a ? ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? ? c     a ? ? ? ? ? ? ? ? ? ? c     a ? ? ? ? ? ? ? ? ? ? c
b ? ? ? ? ? ? ? ? ? ? a     b ? ? ? ? ? ? ? ? ? ? a     b ? ? ? ? ? ? ? ? ? ? a
b ? ? ? ? ? ? ? ? ? ? a     b ? ? ? ? ? ? ? ? ? ? a     b ? ? ? ? ? ? ? ? ? ? a
c ? ? ? ? ? ? ? ? ? ? a     c ? ? ? ? ? ? ? ? ? ? a     c ? ? ? ? ? ? ? ? ? ? a
d ? ? ? ? ? ? ? ? ? ? a     d ? ? ? ? ? ? ? ? ? ? a     d ? ? ? ? ? ? ? ? ? ? a
r ? ? ? ? ? ? ? ? ? ? b     r ? ? ? ? ? ? ? ? ? ? b     r ? ? ? ? ? ? ? ? ? ? b
r ? ? ? ? ? ? ? ? ? ? b     r ? ? ? ? ? ? ? ? ? ? b     r ? ? ? ? ? ? ? ? ? ? b
```

FIGURE 9.10 The three possibilities ("b", "c", or "d") for the third element of the first row of M(*Text*) when BWT(*Text*) is "ard$rcaaaabb". One of these possibilities must correspond to the second symbol of *Text*.

STOP and Think: How would you choose among "b", "c", and "d" for the second symbol of *Text*?

The First-Last Property

To determine the remaining symbols of *Text*, we need to use a subtle property of M(*Text*) that may seem completely unrelated to inverting the Burrows-Wheeler transform. Below, we have indexed the occurrences of each symbol in *FirstColumn* with subscripts according to their order of appearance in this column. When *Text* = "panamabananas$", six instances of "a" appear in *FirstColumn*.

```
$  panamabananas
a1 bananas$panam
a2 mabananas$pan
a3 namabananas$p
a4 nanas$panamab
a5 nas$panamaban
a6 s$panamabanan
b  ananas$panama
m  abananas$pana
n  amabananas$pa
n  anas$panamaba
n  as$panamabana
p  anamabananas$
s  $panamabanana
```

Consider "a_1" in *FirstColumn*, which occurs at the beginning of the cyclic rotation "$a_1$bananas\$panam". If we cyclically rotate this string, then we obtain "panama$a_1$bananas\$". Thus, "a_1" in *FirstColumn* is actually the third occurrence of "a" in "panam**a**bananas\$". We can now identify the positions of the other five instances of "a" in "panamabananas\$":

pa_3na_2ma_1ba_4na_5na_6s\$

EXERCISE BREAK: Where are the three instances of "n" from *FirstColumn* (i.e., "n_1", "n_2", and "n_3") located in "panamabananas\$"?

To locate "a_1" in *LastColumn*, we need to cyclically rotate the second row of the matrix M("$a_1$bananas\$panam"), which results in "bananas\$panamaa_1". This rotation corresponds to the eighth row of M("panamabananas\$"):

```
$  panamabananas
a1 bananas$panam
a2 mabananas$pan
a3 namabananas$p
a4 nanas$panamab
a5 nas$panamaban
a6 s$panamabanan
b  ananas$panama1
m  abananas$pana
n  amabananas$pa
n  anas$panamaba
n  as$panamabana
p  anamabananas$
s  $panamabanana
```


EXERCISE BREAK: Where are the other five instances of "a" located in *LastColumn*?

You hopefully saw that *LastColumn* can be recorded as "smnpbnn$a_1a_2a_3a_4a_5$\$$a_6$", as shown in Figure 9.11. Note that the six instances of "a" appear in exactly the same order in *FirstColumn* and *LastColumn*. This observation is not a fluke. On the contrary, it is a principle that holds for any string *Text* and any symbol that we choose.

First-Last Property: *The k-th occurrence of a symbol in FirstColumn and the k-th occurrence of this symbol in LastColumn correspond to the same position of this symbol in Text.*

```
$  panamabananas
a1 bananas$panam
a2 mabananas$pan
a3 namabananas$p
a4 nanas$panamab
a5 nas$panamaban
a6 s$panamabanan
b  ananas$panama1
m  abananas$pana2
n  amabananas$pa3
n  anas$panamaba4
n  as$panamabana5
p  anamabananas$
s  $panamabanana6
```

FIGURE 9.11 The six occurrences of "a" occur in the same order in *FirstColumn* as they do in *LastColumn*.

To see why the First-Last Property is true, consider the rows of M("panamabananas\$") beginning with "a":

```
a1 bananas$panam
a2 mabananas$pan
a3 namabananas$p
a4 nanas$panamab
a5 nas$panamaban
a6 s$panamabanan
```

These rows are already ordered lexicographically, so if we chop off the "a" from the beginning of each row, then the remaining strings should still be ordered lexicographically:

b a n a n a s \$ p a n a m
m a b a n a n a s \$ p a n
n a m a b a n a n a s \$ p
n a n a s \$ p a n a m a b
n a s \$ p a n a m a b a n
s \$ p a n a m a b a n a n

Adding "a" back to the end of each row should not change the lexicographic ordering of these rows:

b a n a n a s \$ p a n a m a_1
m a b a n a n a s \$ p a n a_2
n a m a b a n a n a s \$ p a_3
n a n a s \$ p a n a m a b a_4
n a s \$ p a n a m a b a n a_5
s \$ p a n a m a b a n a n a_6

But these are just the rows of M("panamabananas\$") containing "a" in *LastColumn*! As a result, the k-th occurrence of "a" in *FirstColumn* corresponds to the k-th occurrence of "a" in *LastColumn*. This argument generalizes for any *symbol* and any string *Text*, which establishes the First-Last property.

Using the First-Last property to invert the Burrows-Wheeler transform

The First-Last Property is interesting, but how can we use it to invert BWT(*Text*) = "ard\$rcaaaabb"? Recalling Figure 9.10, let's return to where we were in our attempt to reconstruct the first row of M(*Text*) and index the occurrences of each symbol in *FirstColumn* and *LastColumn*:

$\$_1$ a ? ? ? ? ? ? ? ? ? ? ? a_1
a_1 ? ? ? ? ? ? ? ? ? ? ? ? r_1
a_2 ? ? ? ? ? ? ? ? ? ? ? ? d_1
a_3 ? ? ? ? ? ? ? ? ? ? ? ? $\$_1$
a_4 ? ? ? ? ? ? ? ? ? ? ? ? r_2
a_5 ? ? ? ? ? ? ? ? ? ? ? ? c_1
b_1 ? ? ? ? ? ? ? ? ? ? ? ? a_2
b_2 ? ? ? ? ? ? ? ? ? ? ? ? a_3
c_1 ? ? ? ? ? ? ? ? ? ? ? ? a_4
d_1 ? ? ? ? ? ? ? ? ? ? ? ? a_5
r_1 ? ? ? ? ? ? ? ? ? ? ? ? b_1
r_2 ? ? ? ? ? ? ? ? ? ? ? ? b_2

The First-Last Property reveals where "a_3" is hiding in *LastColumn*:

$\$_1$ a ? ? ? ? ? ? ? ? ? ? ? a_1
a_1 ? ? ? ? ? ? ? ? ? ? ? ? r_1
a_2 ? ? ? ? ? ? ? ? ? ? ? ? d_1
a_3 ? ? ? ? ? ? ? ? ? ? ? ? $\$_1$
a_4 ? ? ? ? ? ? ? ? ? ? ? ? r_2
a_5 ? ? ? ? ? ? ? ? ? ? ? ? c_1
b_1 ? ? ? ? ? ? ? ? ? ? ? ? a_2
b_2 ? ? ? ? ? ? ? ? ? ? ? ? a_3
c_1 ? ? ? ? ? ? ? ? ? ? ? ? a_4
d_1 ? ? ? ? ? ? ? ? ? ? ? ? a_5
r_1 ? ? ? ? ? ? ? ? ? ? ? ? b_1
r_2 ? ? ? ? ? ? ? ? ? ? ? ? b_2

Since we know that "a_3" is located at the end of the eighth row, we can wrap around this row to determine that "b_2" follows "a_3" in *Text*. Thus, the second symbol of *Text* is "b", which we can now add to the first row of M(*Text*):

$\$_1$ a b ? ? ? ? ? ? ? ? ? ? a_1
a_1 ? ? ? ? ? ? ? ? ? ? ? ? r_1
a_2 ? ? ? ? ? ? ? ? ? ? ? ? d_1
a_3 ? ? ? ? ? ? ? ? ? ? ? ? $\$_1$
a_4 ? ? ? ? ? ? ? ? ? ? ? ? r_2
a_5 ? ? ? ? ? ? ? ? ? ? ? ? c_1
b_1 ? ? ? ? ? ? ? ? ? ? ? ? a_2
b_2 ? ? ? ? ? ? ? ? ? ? ? ? a_3
c_1 ? ? ? ? ? ? ? ? ? ? ? ? a_4
d_1 ? ? ? ? ? ? ? ? ? ? ? ? a_5
r_1 ? ? ? ? ? ? ? ? ? ? ? ? b_1
r_2 ? ? ? ? ? ? ? ? ? ? ? ? b_2

In Figure 9.12, we illustrate repeated applications of the First-Last Property to reconstruct more and more symbols from *Text*. Presto — the string that we have been trying to reconstruct is "abracadabra\$".

EXERCISE BREAK: Reconstruct the string whose Burrows-Wheeler transform is "enwvpeoseu\$llt".

STOP and Think: Can *any* string (having a single "\$" symbol) be inverted using the inverse Burrows-Wheeler transform?

You are now ready to implement the inverse of the Burrows-Wheeler transform.

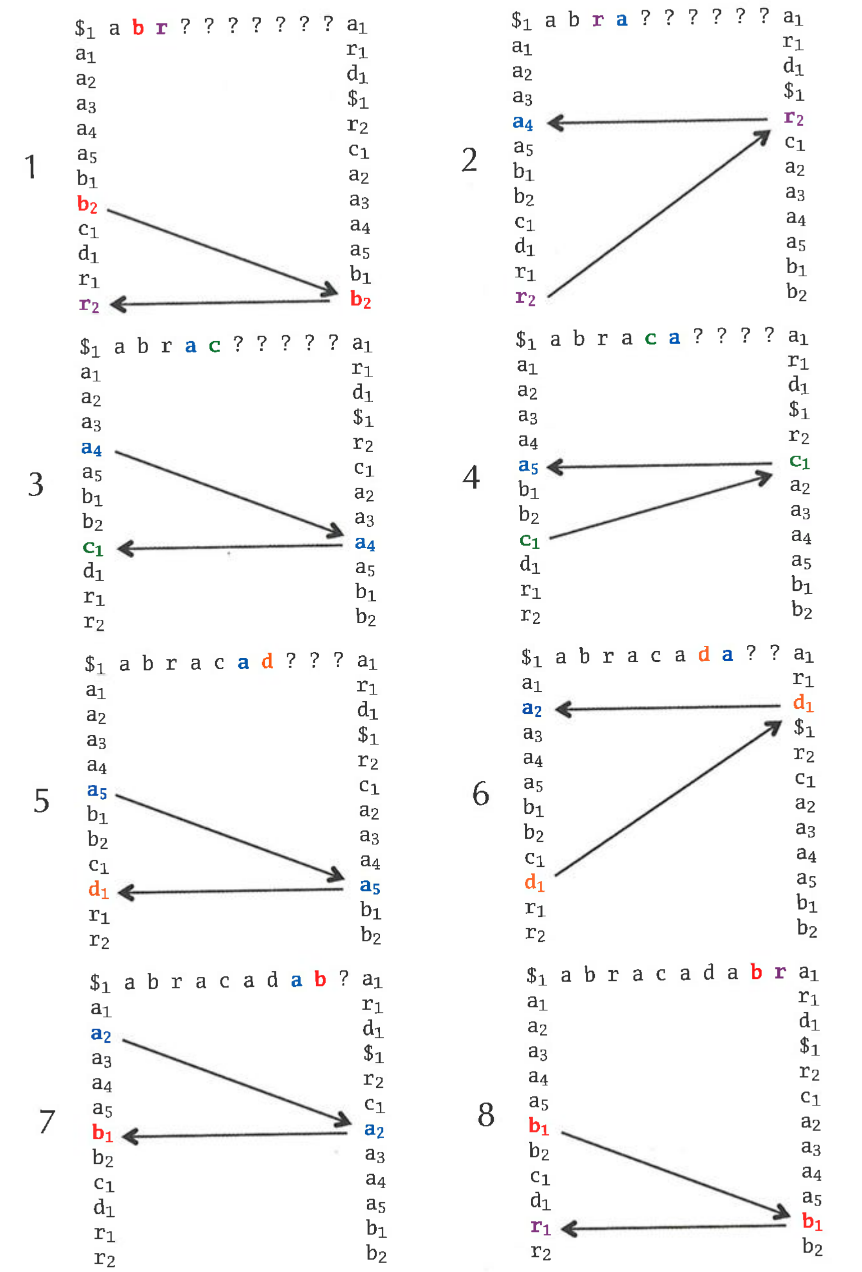


FIGURE 9.12 Repeated applications of the First-Last Property reconstruct the string "abracadabra\$" from its Burrows-Wheeler transform "ard\$rcaaaabb".

9J

Inverse Burrows-Wheeler Transform Problem:
Reconstruct a string from its Burrows-Wheeler transform.

Input: A string *Transform* (with a single "\$" symbol).
Output: The string *Text* such that BWT(*Text*) = *Transform*.

Pattern Matching with the Burrows-Wheeler Transform

A first attempt at Burrows-Wheeler pattern matching

The Burrows-Wheeler transform may be fascinating, but how could it possibly help us decrease the memory required for pattern matching? The idea motivating a Burrows-Wheeler-based approach to pattern matching relies on the observation that each row of M(*Text*) begins with a different suffix of *Text*. Since these suffixes are already ordered lexicographically, as we already noted when pattern matching with the suffix array, any matches of *Pattern* in *Text* will appear at the beginning of consecutive rows of M(*Text*), as shown in Figure 9.13.

M(Text)	SUFFIXARRAY(*Text*)
\$ p a n a m a b a n a n a s	13
a b a n a n a s \$ p a n a m	5
a m a b a n a n a s \$ p a n	3
a n a m a b a n a n a s \$ p	**1**
a n a n a s \$ p a n a m a b	**7**
a n a s \$ p a n a m a b a n	**9**
a s \$ p a n a m a b a n a n	11
b a n a n a s \$ p a n a m a	6
m a b a n a n a s \$ p a n a	4
n a m a b a n a n a s \$ p a	2
n a n a s \$ p a n a m a b a	8
n a s \$ p a n a m a b a n a	10
p a n a m a b a n a n a s \$	0
s \$ p a n a m a b a n a n a	12

FIGURE 9.13 (Left) Because the rows of M(*Text*) are ordered lexicographically, suffixes beginning with the same string ("**ana**") appear in consecutive rows of the matrix. (Right) The suffix array records the starting position of each suffix in *Text* and immediately tells us the locations of "**ana**".

have the outline of a method to match *Pattern* to *Text*. Construct M(*Text*), entify rows beginning with the first symbol of *Pattern*. Among these rows, which ones have a second element matching the second symbol of *Pattern*. his process until we find which rows of M(*Text*) begin with *Pattern*.

nd Think: What is wrong with this approach?

kward through a pattern

vith this proposed method for pattern matching is that we cannot afford entire matrix M(*Text*), which has $|Text|^2$ entries. In an effort to reduce mem- ements, let's forbid ourselves from accessing any information in M(*Text*) *FirstColumn* and *LastColumn*. Using these two columns, we will try to ern to *Text* by moving *backward* through *Pattern*. For example, if we want to ern = "ana" to *Text* = "panamabananas$", then we will first identify rows of ginning with "a", the last letter of "ana":

```
$1  p a n a m a b a n a n a  s1
a1  b a n a n a s $ p a n a  m1
a2  m a b a n a n a s $ p a  n1
a3  n a m a b a n a n a s $  p1
a4  n a n a s $ p a n a m a  b1
a5  n a s $ p a n a m a b a  n2
a6  s $ p a n a m a b a n a  n3
b1  a n a n a s $ p a n a m  a1
m1  a b a n a n a s $ p a n  a2
n1  a m a b a n a n a s $ p  a3
n2  a n a s $ p a n a m a b  a4
n3  a s $ p a n a m a b a n  a5
p1  a n a m a b a n a n a s  $1
s1  $ p a n a m a b a n a n  a6
```

are moving backward through "ana", we will next look for rows of M(*Text*) with "na". To do this without knowing the entire matrix M(*Text*), we again ct that a symbol in *LastColumn* must precede the symbol of *Text* found in row in *FirstColumn*. Thus, we only need to identify those rows of M(*Text*) with "a" and ending with "n":


```
$1  p a n a m a b a n a n a  s1
a1  b a n a n a s $ p a n a  m1
a2  m a b a n a n a s $ p a  n1
a3  n a m a b a n a n a s $  p1
a4  n a n a s $ p a n a m a  b1
a5  n a s $ p a n a m a b a  n2
a6  s $ p a n a m a b a n a  n3
b1  a n a n a s $ p a n a m  a1
m1  a b a n a n a s $ p a n  a2
n1  a m a b a n a n a s $ p  a3
n2  a n a s $ p a n a m a b  a4
n3  a s $ p a n a m a b a n  a5
p1  a n a m a b a n a n a s  $1
s1  $ p a n a m a b a n a n  a6
```

The First-Last Property tells us where to find the three highlighted "n" in *FirstColumn*, as shown below. All three rows end with "a", yielding three total occurrences of "ana" in *Text*.

```
$1  p a n a m a b a n a n a  s1
a1  b a n a n a s $ p a n a  m1
a2  m a b a n a n a s $ p a  n1
a3  n a m a b a n a n a s $  p1
a4  n a n a s $ p a n a m a  b1
a5  n a s $ p a n a m a b a  n2
a6  s $ p a n a m a b a n a  n3
b1  a n a n a s $ p a n a m  a1
m1  a b a n a n a s $ p a n  a2
n1  a m a b a n a n a s $ p  a3
n2  a n a s $ p a n a m a b  a4
n3  a s $ p a n a m a b a n  a5
p1  a n a m a b a n a n a s  $1
s1  $ p a n a m a b a n a n  a6
```

The highlighted occurrences of "a" in *LastColumn* correspond to the third, fourth, and fifth occurrences of "a" in this column, and the First-Last Property tells us that they should correspond to the third, fourth, and fifth occurrences of "a" in *FirstColumn* as well, which identifies the three matches of "ana":

```
$1 p a n a m a b a n a n a s1
a1 b a n a n a s $ p a n a m1
a2 m a b a n a n a s $ p a n1
a3 n a m a b a n a n a s $ p1
a4 n a n a s $ p a n a m a b1
a5 n a s $ p a n a m a b a n2
a6 s $ p a n a m a b a n a n3
b1 a n a n a s $ p a n a m a1
m1 a b a n a n a s $ p a n a2
n1 a m a b a n a n a s $ p a3
n2 a n a s $ p a n a m a b a4
n3 a s $ p a n a m a b a n a5
p1 a n a m a b a n a n a s $1
s1 $ p a n a m a b a n a n a6
```

EXERCISE BREAK: Match *Pattern* = "banana" to *Text* = "panamabananas$" by walking backward through *Pattern* using the Burrows-Wheeler transform of *Text*.

The Last-to-First mapping

We now know how to use BWT(*Text*) to find all matches of *Pattern* in *Text* by walking backward through *Pattern*. However, every time we walk backward, we need to keep track of the rows of M(*Text*) where the matches of a suffix of *Pattern* are hiding. Fortunately, we know that at each step, the rows of M(*Text*) that match a suffix of *Pattern* clump together in consecutive rows of M(*Text*). This means that the collection of all matching rows is revealed by only two pointers, *top* and *bottom*: *top* holds the index of the first row of M(*Text*) that matches the current suffix of *Pattern*, and *bottom* holds the index of the last row of M(*Text*) that matches this suffix. Figure 9.14 shows the process of updating pointers; after walking backward through *Pattern* = "ana", we have that $top = 3$ and $bottom = 5$. After traversing *Pattern*, we can compute the total number of matches of *Pattern* in *Text* by calculating $bottom - top + 1$ (e.g., there are $5 - 3 + 1 = 3$ matches of "ana" in "panamabananas$").

Let's concentrate on how pointers are updated from one stage to the next. Consider the transition from the second to the third panel in Figure 9.14; how did we know to update the pointers ($top = 1$, $bottom = 6$) into ($top = 9$, $bottom = 11$)? We are looking for the first and last occurrence of "n" in the range of positions from $top = 1$ to $bottom = 6$ in *LastColumn*. The first occurrence of "n" in this range is "n_1" (in position 2) and the last is "n_3" (position 6).

In order to update the *top* and *bottom* pointers, we need to determine where "n_1" and "n_3" occur in *FirstColumn*. The **Last-to-First mapping**, denoted LASTTOFIRST(i),

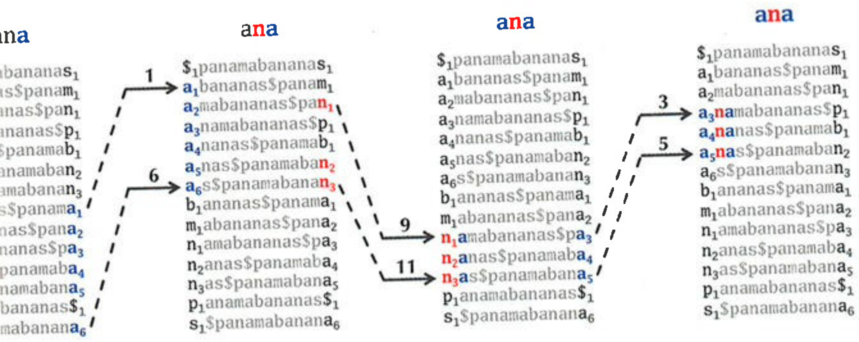
answers the following question: given a symbol at position i in *LastColumn*, what is its position in *FirstColumn*? 9K

For our ongoing example, LASTTOFIRST(2) = 9, since the symbol at position 2 of *LastColumn* ("n_1") occurs at position 9 in *FirstColumn*, as shown in Figure 9.15. Similarly, LASTTOFIRST(6) = 11, since the symbol at position 6 of *LastColumn* ("n_3") occurs at position 11 in *FirstColumn*. Therefore, with the help of the Last-to-First mapping, we can quickly update the pointers ($top = 1$, $bottom = 6$) into ($top = 9$, $bottom = 11$).

We are now ready to describe **BWMATCHING**, an algorithm that counts the total number of matches of *Pattern* in *Text*, where the only information that we are given is *FirstColumn* and *LastColumn* in addition to the Last-to-First mapping. The pointers *top* and *bottom* are updated by the green lines in the following pseudocode.

```
BWMATCHING(FirstColumn, LastColumn, Pattern, LASTTOFIRST)
    top ← 0
    bottom ← |LastColumn| − 1
    while top ≤ bottom
        if Pattern is nonempty
            symbol ← last letter in Pattern
            remove last letter from Pattern
            if positions from top to bottom in LastColumn contain symbol
                topIndex ← first position of symbol among positions from top to bottom
                        in LastColumn
                bottomIndex ← last position of symbol among positions from top to
                        bottom in LastColumn
                top ← LASTTOFIRST(topIndex)
                bottom ← LASTTOFIRST(bottomIndex)
            else
                return 0
        else
            return bottom − top + 1
```

9L



E 9.14 The pointers *top* and *bottom* hold the indices of the first and last rows *Text*) matching the current suffix of *Pattern* = "ana". The above diagram shows hese pointers are updated when walking backwards through "ana" and looking ostring matches in "panamabananas$".

FirstColumn	*LastColumn*	LASTTOFIRST(i)	COUNT $	a	b	m	n	p	s
$\$_1$	s_1	13	0	0	0	0	0	0	0
a_1	m_1	8	0	0	0	0	0	0	1
a_2	n_1	9	0	0	0	1	0	0	1
a_3	p_1	12	0	0	0	1	1	0	1
a_4	b_1	7	0	0	0	1	1	1	1
a_5	n_2	10	0	0	1	1	1	1	1
a_6	n_3	11	0	0	1	1	2	1	1
b_1	a_1	1	0	0	1	1	3	1	1
m_1	a_2	2	0	1	1	1	3	1	1
n_1	a_3	3	0	2	1	1	3	1	1
n_2	a_4	4	0	3	1	1	3	1	1
n_3	a_5	5	0	4	1	1	3	1	1
p_1	$\$_1$	0	0	5	1	1	3	1	1
s_1	a_6	6	1	5	1	1	3	1	1
			1	6	1	1	3	1	1

URE 9.15 The Last-to-First mapping and count array. Precomputing the count array ents time-consuming updates of the *top* and *bottom* pointers in **BWMATCHING**.

Speeding Up Burrows-Wheeler Pattern Matching

Substituting the Last-to-First mapping with count arrays

If you implemented **BWMATCHING** in the previous section, you probably found this algorithm to be slow. The reason for its sluggishness is that updating the pointers *top* and *bottom* is time-intensive, since it requires examining every symbol in *LastColumn* between *top* and *bottom* at each step. To improve **BWMATCHING**, we introduce a function $\text{COUNT}_{symbol}(i, LastColumn)$, which returns the number of occurrences of *symbol* in the first *i* positions of *LastColumn*. For example, $\text{COUNT}_{\text{"n"}}(10, \text{"smnpbnnaaaaa\$a"}) = 3$, and $\text{COUNT}_{\text{"a"}}(4, \text{"smnpbnnaaaaa\$a"}) = 0$. In Figure 9.15, we show arrays holding $\text{COUNT}_{symbol}(i, \text{"smnpbnnaaaaa\$a"})$ for every *symbol* occurring in "panamabananas$".

EXERCISE BREAK: Compute the arrays COUNT for BWT("abracadabra$").

We will say that the *k*-th occurrence of *symbol* in a column of a matrix has **rank** *k* in this column. For *Text* = "panamabananas$", note that the first and last occurrences of *symbol* in the range of positions from *top* to *bottom* in *LastColumn* have respective ranks

$$\text{COUNT}_{symbol}(top, LastColumn) + 1$$

and

$$\text{COUNT}_{symbol}(bottom + 1, LastColumn).$$

As illustrated in Figure 9.15, when *top* = 1, *bottom* = 6, and *symbol* = "n",

$$\text{COUNT}_{\text{"n"}}(top, LastColumn) + 1 = 1$$
$$\text{COUNT}_{\text{"n"}}(bottom + 1, LastColumn) = 3$$

The occurrences of "n" having ranks 1 and 3 are located at positions 2 and 6 of *LastColumn*, implying that we should update *top* to LASTTOFIRST(2) = 9 and *bottom* to LASTTOFIRST(6) = 11. Thus, the four green lines in the pseudocode for **BWMATCHING** can be rewritten as follows.

```
topIndex ← position of symbol with rank COUNT_symbol(top, LastColumn) + 1
        in LastColumn
bottomIndex ← position of symbol with rank COUNT_symbol(bottom + 1, LastColumn)
        in LastColumn
top ← LASTTOFIRST(topIndex)
bottom ← LASTTOFIRST(bottomIndex)
```