The Burrows-Wheeler Transform

Solon P. Pissis Hilde Verbeek

CWI & VU, Amsterdam

11 November 2024

Burrows-Wheeler Transform (BWT)

- We will consider a string $T = T[0\,..\,n-1]$ of length n over alphabet Σ.
  - e.g. a DNA sequence is a string over $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$
- The *Burrows-Wheeler Transform* (**BWT**) is a way of permuting the letters of T into another string $\mathrm{BWT}(T)$.

Example

Let $T = \mathtt{banana\$}$. Then $\mathrm{BWT}(T) = \mathtt{annb\$aa}$.

Burrows-Wheeler Transform (BWT)

- The technique was first described by Michael Burrows and David Wheeler in 1994 [1].
- Later works have greatly improved on its complexity and applications.
- The BWT has two main (direct!) applications:
  - compression (e.g. `bzip2`); and
  - indexing (e.g. `BWA` [6] and `Bowtie` [5]).

Today's lecture

1. Computation and compression
2. Reversal
3. Pattern matching

Today's lecture

1. Computation and compression
2. Reversal
3. Pattern matching

Computing the BWT

$$T = \text{banana\$}$$

Given a string T of length n, the BWT can be computed as follows:

1. Compute all rotations of T
2. Sort the rotations lexicographically
3. Take the final letter of each rotation and concatenate

We always add a unique \$ to the end of the string (\$ being smaller than all other letters in the alphabet). This is necessary for reversing the BWT as we will see later.

Computing the BWT

$$T = \texttt{banana\$}$$

The rotations of a string are obtained by shifting all letters forward, moving the first letters towards the end:

```
banana$
anana$b
nana$ba
ana$ban
na$bana
a$banan
$banana
```

Computing the BWT

$$T = \mathtt{banana\$}$$

We sort all rotations (including T itself) lexicographically.

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

Computing the BWT

$$T = \texttt{banana\$}$$

Finally, we take the last letter of each rotation and concatenate to obtain the BWT.

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

$$\mathrm{BWT}(T) = \texttt{annb\$aa}$$

Run-length encoding

- Run-length encoding (RLE) is a simple compression scheme that tends to work well in tandem with BWT.
- Every run of the same character repeating is encoded as said character plus the length of the run.
  - e.g. `AAAAAAA` becomes `A7`
- Both encoding and decoding can be done in linear time.

Example

$$T = \texttt{TTTTTGGGACCTTTG}$$
$$\mathrm{RLE}(T) = \texttt{T5G3A1C2T3G1}$$

RLE and BWT

Consider this long string containing `hello` several times:

$$T = \ldots\texttt{hello}\ldots\ \ldots\texttt{hello}\ldots\ \ldots\texttt{hello}\ldots\ \ldots\texttt{hello}\ldots\ \$$$

The rotations starting at equivalent positions within `hello` will likely appear consecutively after sorting:

```
 ⋮      ⋮
ello…   h
ello…   h
ello…   h
ello…   h
 ⋮      ⋮
llo…    e
llo…    e
llo…    e
llo…    e
 ⋮      ⋮
```

$$\mathrm{BWT}(T) = \ldots\texttt{hhhh}\ldots\ \ldots\texttt{eeee}\ldots$$
$$\ldots\texttt{llll}\ldots\ \ldots\texttt{llll}\ldots$$

Why compression works

Thus, repeating patterns are often turned into several long runs (but similar phrases, like `yellow`, may throw a wrench in this).

Today's lecture

Reversing BWT

- BWT is reversible, using an important property called the *LF* Mapping.
- The main idea is that we rank all occurrences of each letter, and use a correspondence between these ranks to spell the original string backwards.

Burrows-Wheeler Matrix

Let $T = \texttt{abaaba\$}$. We take T's rotations and sort them, storing the first and last letters of each in the following table:

F		L
\$	abaab	a
a	\$abaa	b
a	aba\$a	b
a	ba\$ab	a
a	baaba	\$
b	a\$aba	a
b	aaba\$	a

This is called the Burrows-Wheeler Matrix (BWM). The F column contains all letters sorted, and the L column spells the BWT.

LF mapping

We rank the letters of T: the first a becomes a_0, the second a becomes a_1, and so on:

$$T = a_0 b_0 a_1 a_2 b_1 a_3 \$$$
$$\mathrm{BWT}(T) = a_3 b_1 b_0 a_1 \$ a_2 a_0$$

F						L
\$	a_0	b_0	a_1	a_2	b_1	a_3
a_3	\$	a_0	b_0	a_1	a_2	b_1
a_1	a_2	b_1	a_3	\$	a_0	b_0
a_2	b_1	a_3	\$	a_0	b_0	a_1
a_0	b_0	a_1	a_2	b_1	a_3	\$
b_1	a_3	\$	a_0	b_0	a_1	a_2
b_0	a_1	a_2	b_1	a_3	\$	a_0

The ranks for each letter appear in the same order in both F and L columns. This property is called LF mapping.

LF mapping

We rank the letters of T: the first a becomes $\mathtt{a}_0$, the second a becomes $\mathtt{a}_1$, and so on:

$$T = \mathtt{a}_0\mathtt{b}_0\mathtt{a}_1\mathtt{a}_2\mathtt{b}_1\mathtt{a}_3\$$$
$$\mathrm{BWT}(T) = \mathtt{a}_3\mathtt{b}_1\mathtt{b}_0\mathtt{a}_1\$\mathtt{a}_2\mathtt{a}_0$$

F						L
\$	$\mathtt{a}_0$	$\mathtt{b}_0$	$\mathtt{a}_1$	$\mathtt{a}_2$	$\mathtt{b}_1$	$\mathtt{a}_3$
$\mathtt{a}_3$	\$	$\mathtt{a}_0$	$\mathtt{b}_0$	$\mathtt{a}_1$	$\mathtt{a}_2$	$\mathtt{b}_1$
$\mathtt{a}_1$	$\mathtt{a}_2$	$\mathtt{b}_1$	$\mathtt{a}_3$	\$	$\mathtt{a}_0$	$\mathtt{b}_0$
$\mathtt{a}_2$	$\mathtt{b}_1$	$\mathtt{a}_3$	\$	$\mathtt{a}_0$	$\mathtt{b}_0$	$\mathtt{a}_1$
$\mathtt{a}_0$	$\mathtt{b}_0$	$\mathtt{a}_1$	$\mathtt{a}_2$	$\mathtt{b}_1$	$\mathtt{a}_3$	\$
$\mathtt{b}_1$	$\mathtt{a}_3$	\$	$\mathtt{a}_0$	$\mathtt{b}_0$	$\mathtt{a}_1$	$\mathtt{a}_2$
$\mathtt{b}_0$	$\mathtt{a}_1$	$\mathtt{a}_2$	$\mathtt{b}_1$	$\mathtt{a}_3$	\$	$\mathtt{a}_0$

The ranks for each letter appear in the same order in both F and L columns. This property is called LF mapping.

LF mapping

We rank the letters of T: the first a becomes $\mathtt{a}_0$, the second a becomes $\mathtt{a}_1$, and so on:

$$T = \mathtt{a}_0\mathtt{b}_0\mathtt{a}_1\mathtt{a}_2\mathtt{b}_1\mathtt{a}_3\$$$
$$\mathrm{BWT}(T) = \mathtt{a}_3\mathtt{b}_1\mathtt{b}_0\mathtt{a}_1\$\mathtt{a}_2\mathtt{a}_0$$

F						L
\$	a_0	b_0	a_1	a_2	b_1	a_3
a_3	\$	a_0	b_0	a_1	a_2	b_1
a_1	a_2	b_1	a_3	\$	a_0	b_0
a_2	b_1	a_3	\$	a_0	b_0	a_1
a_0	b_0	a_1	a_2	b_1	a_3	\$
b_1	a_3	\$	a_0	b_0	a_1	a_2
b_0	a_1	a_2	b_1	a_3	\$	a_0

The ranks for each letter appear in the same order in both F and L columns. This property is called LF mapping.

LF mapping

We rank the letters of T: the first a becomes a_0, the second a becomes a_1, and so on:

$$T = a_0 b_0 a_1 a_2 b_1 a_3 \$$$

$$\mathrm{BWT}(T) = a_3 b_1 b_0 a_1 \$ a_2 a_0$$

F						L
\$	a_0	b_0	a_1	a_2	b_1	a_3
a_3	\$	a_0	b_0	a_1	a_2	b_1
a_1	a_2	b_1	a_3	\$	a_0	b_0
a_2	b_1	a_3	\$	a_0	b_0	a_1
a_0	b_0	a_1	a_2	b_1	a_3	\$
b_1	a_3	\$	a_0	b_0	a_1	a_2
b_0	a_1	a_2	b_1	a_3	\$	a_0

Consider all rotations starting with a. These will occur consecutively in the BWM, and be sorted by whatever comes *after* the initial a.

LF mapping

We rank the letters of T: the first a becomes a_0, the second a becomes a_1, and so on:

$$T = a_0 b_0 a_1 a_2 b_1 a_3 \$$$
$$\mathrm{BWT}(T) = a_3 b_1 b_0 a_1 \$ a_2 a_0$$

F						L
$\$$	a_0	b_0	a_1	a_2	b_1	a_3
a_3	$\$$	a_0	b_0	a_1	a_2	b_1
a_1	a_2	b_1	a_3	$\$$	a_0	b_0
a_2	b_1	a_3	$\$$	a_0	b_0	a_1
a_0	b_0	a_1	a_2	b_1	a_3	$\$$
b_1	a_3	$\$$	a_0	b_0	a_1	a_2
b_0	a_1	a_2	b_1	a_3	$\$$	a_0

The rotations *ending* with a, while not appearing consecutively, are also sorted by what comes directly after the a.

LF mapping

We rank the letters of T: the first a becomes a_0, the second a becomes a_1, and so on:

$$T = a_0 b_0 a_1 a_2 b_1 a_3 \$$$

$$\mathrm{BWT}(T) = a_3 b_1 b_0 a_1 \$ a_2 a_0$$

F						L
\$	a_0	b_0	a_1	a_2	b_1	a_3
a_3	\$	a_0	b_0	a_1	a_2	b_1
a_1	a_2	b_1	a_3	\$	a_0	b_0
a_2	b_1	a_3	\$	a_0	b_0	a_1
a_0	b_0	a_1	a_2	b_1	a_3	\$
b_1	a_3	\$	a_0	b_0	a_1	a_2
b_0	a_1	a_2	b_1	a_3	\$	a_0

We use the LF mapping to determine which letter precedes a given other letter.

LF mapping

We rank the letters of T: the first a becomes a_0, the second a becomes a_1, and so on:

$$T = a_0 b_0 a_1 a_2 b_1 a_3 \$$$
$$\mathrm{BWT}(T) = a_3 b_1 b_0 a_1 \$ a_2 a_0$$

F						L
\$	a_0	b_0	a_1	a_2	b_1	a_3
a_3	\$	a_0	b_0	a_1	a_2	b_1
a_1	a_2	b_1	a_3	\$	a_0	b_0
a_2	b_1	a_3	\$	a_0	b_0	a_1
a_0	b_0	a_1	a_2	b_1	a_3	\$
b_1	a_3	\$	a_0	b_0	a_1	a_2
b_0	a_1	a_2	b_1	a_3	\$	a_0

For example, to find the letter preceding the second a, we look up the row with a_1 in the F column, and take the letter and rank from the last column.

LF mapping

If we do not know the original sequence, we cannot rank the letters based on it. Luckily, the LF mapping still works when we rank based on occurrences in the BWT.

F	L
\$	a_3
a_3	b_1
a_1	b_0
a_2	a_1
a_0	\$
b_1	a_2
b_0	a_0

T-rank

F	L
\$	a_0
a_0	b_0
a_1	b_1
a_2	a_1
a_3	\$
b_0	a_2
b_1	a_3

B-rank

Example

$$T = \texttt{-----------\$}$$

$$\mathrm{BWT}(T) = \texttt{ipssm\$pissii}$$

We will start by creating the BWM.

Example

$$T = \texttt{-----------\$}$$

$$\text{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
	i
	p
	s
	s
	m
	\$
	p
	i
	s
	s
	i
	i

The L column is taken directly from the BWT.

Example

$$T = \texttt{-----------\$}$$

$$\mathrm{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
\$	i
i	p
i	s
i	s
i	m
m	\$
p	p
p	i
s	s
s	s
s	i
s	i

The F column is obtained by sorting all letters in the BWT.

Example

$$T = \texttt{-----------\$}$$

$$\mathrm{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
\$	$\mathtt{i}_0$
$\mathtt{i}_0$	$\mathtt{p}_0$
$\mathtt{i}_1$	$\mathtt{s}_0$
$\mathtt{i}_2$	$\mathtt{s}_1$
$\mathtt{i}_3$	$\mathtt{m}_0$
$\mathtt{m}_0$	\$
$\mathtt{p}_0$	$\mathtt{p}_1$
$\mathtt{p}_1$	$\mathtt{i}_1$
$\mathtt{s}_0$	$\mathtt{s}_2$
$\mathtt{s}_1$	$\mathtt{s}_3$
$\mathtt{s}_2$	$\mathtt{i}_2$
$\mathtt{s}_3$	$\mathtt{i}_3$

Next, we rank the letters in both columns, using B-rank.

Example

$$T = \texttt{-----------\$}$$

$$\text{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
$\$$	$\texttt{i}_0$
$\texttt{i}_0$	$\texttt{p}_0$
$\texttt{i}_1$	$\texttt{s}_0$
$\texttt{i}_2$	$\texttt{s}_1$
$\texttt{i}_3$	$\texttt{m}_0$
$\texttt{m}_0$	$\$$
$\texttt{p}_0$	$\texttt{p}_1$
$\texttt{p}_1$	$\texttt{i}_1$
$\texttt{s}_0$	$\texttt{s}_2$
$\texttt{s}_1$	$\texttt{s}_3$
$\texttt{s}_2$	$\texttt{i}_2$
$\texttt{s}_3$	$\texttt{i}_3$

We can now start spelling the string backwards. We already know it ends with $\$$.

Example

$$T = \texttt{----------i\$}$$

$$\text{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
$\$$	$\texttt{i}_0$
$\texttt{i}_0$	$\texttt{p}_0$
$\texttt{i}_1$	$\texttt{s}_0$
$\texttt{i}_2$	$\texttt{s}_1$
$\texttt{i}_3$	$\texttt{m}_0$
$\texttt{m}_0$	$\$$
$\texttt{p}_0$	$\texttt{p}_1$
$\texttt{p}_1$	$\texttt{i}_1$
$\texttt{s}_0$	$\texttt{s}_2$
$\texttt{s}_1$	$\texttt{s}_3$
$\texttt{s}_2$	$\texttt{i}_2$
$\texttt{s}_3$	$\texttt{i}_3$

Remember the BWM is made of suffix+prefix

We look up the previous letter in the F column and take the corresponding letter from the L column.

Example

$$T = \text{---------pi\$}$$

$$\text{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
\$	$\texttt{i}_0$
$\texttt{i}_0$	$\texttt{p}_0$
$\texttt{i}_1$	$\texttt{s}_0$
$\texttt{i}_2$	$\texttt{s}_1$
$\texttt{i}_3$	$\texttt{m}_0$
$\texttt{m}_0$	\$
$\texttt{p}_0$	$\texttt{p}_1$
$\texttt{p}_1$	$\texttt{i}_1$
$\texttt{s}_0$	$\texttt{s}_2$
$\texttt{s}_1$	$\texttt{s}_3$
$\texttt{s}_2$	$\texttt{i}_2$
$\texttt{s}_3$	$\texttt{i}_3$

We look up the previous letter in the F column and take the corresponding letter from the L column.

Example

$$T = \texttt{--------ppi\$}$$

$$\mathrm{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
\$	i_0
i_0	p_0
i_1	s_0
i_2	s_1
i_3	m_0
m_0	\$
p_0	p_1
p_1	i_1
s_0	s_2
s_1	s_3
s_2	i_2
s_3	i_3

We look up the previous letter in the F column and take the corresponding letter from the L column.

Example

$$T = \texttt{-------}\textcolor{red}{\texttt{i}}\texttt{ppi\$}$$

$$\mathrm{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
$\$$	$\texttt{i}_0$
$\texttt{i}_0$	$\texttt{p}_0$
$\texttt{i}_1$	$\texttt{s}_0$
$\texttt{i}_2$	$\texttt{s}_1$
$\texttt{i}_3$	$\texttt{m}_0$
$\texttt{m}_0$	$\$$
$\texttt{p}_0$	$\texttt{p}_1$
$\texttt{p}_1$	$\texttt{i}_1$
$\texttt{s}_0$	$\texttt{s}_2$
$\texttt{s}_1$	$\texttt{s}_3$
$\texttt{s}_2$	$\texttt{i}_2$
$\texttt{s}_3$	$\texttt{i}_3$

We look up the previous letter in the F column and take the corresponding letter from the L column.

Example

$$T = \texttt{------sippi\$}$$

$$\mathrm{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
$	i_0
i_0	p_0
i_1	s_0
i_2	s_1
i_3	m_0
m_0	$
p_0	p_1
p_1	i_1
s_0	s_2
s_1	s_3
s_2	i_2
s_3	i_3

We look up the previous letter in the F column and take the corresponding letter from the L column.

Example

$$T = \texttt{-----}\textcolor{red}{\texttt{s}}\texttt{sippi\$}$$

$$\mathrm{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
\$	$\mathtt{i}_0$
$\mathtt{i}_0$	$\mathtt{p}_0$
$\mathtt{i}_1$	$\mathtt{s}_0$
$\mathtt{i}_2$	$\mathtt{s}_1$
$\mathtt{i}_3$	$\mathtt{m}_0$
$\mathtt{m}_0$	\$
$\mathtt{p}_0$	$\mathtt{p}_1$
$\mathtt{p}_1$	$\mathtt{i}_1$
$\mathtt{s}_0$	$\mathtt{s}_2$
$\mathtt{s}_1$	$\mathtt{s}_3$
$\mathtt{s}_2$	$\mathtt{i}_2$
$\mathtt{s}_3$	$\mathtt{i}_3$

We look up the previous letter in the F column and take the corresponding letter from the L column.

Example

$$T = \texttt{----issippi\$}$$
$$\mathrm{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
$	i_0
i_0	p_0
i_1	s_0
i_2	s_1
i_3	m_0
m_0	$
p_0	p_1
p_1	i_1
s_0	s_2
s_1	s_3
s_2	i_2
s_3	i_3

We look up the previous letter in the F column and take the corresponding letter from the L column.

Example

$$T = \texttt{---sissippi\$}$$
$$\mathrm{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
\$	$\texttt{i}_0$
$\texttt{i}_0$	$\texttt{p}_0$
$\texttt{i}_1$	$\texttt{s}_0$
$\texttt{i}_2$	$\texttt{s}_1$
$\texttt{i}_3$	$\texttt{m}_0$
$\texttt{m}_0$	\$
$\texttt{p}_0$	$\texttt{p}_1$
$\texttt{p}_1$	$\texttt{i}_1$
$\texttt{s}_0$	$\texttt{s}_2$
$\texttt{s}_1$	$\texttt{s}_3$
$\texttt{s}_2$	$\texttt{i}_2$
$\texttt{s}_3$	$\texttt{i}_3$

We look up the previous letter in the F column and take the corresponding letter from the L column.

Example

$$T = \texttt{--ssissippi\$}$$
$$\mathrm{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
$	i_0
i_0	p_0
i_1	s_0
i_2	s_1
i_3	m_0
m_0	$
p_0	p_1
p_1	i_1
s_0	s_2
s_1	s_3
s_2	i_2
s_3	i_3

We look up the previous letter in the F column and take the corresponding letter from the L column.

Example

$$T = \texttt{-issississippi\$}$$

$$\text{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
\$	$\texttt{i}_0$
$\texttt{i}_0$	$\texttt{p}_0$
$\texttt{i}_1$	$\texttt{s}_0$
$\texttt{i}_2$	$\texttt{s}_1$
$\texttt{i}_3$	$\texttt{m}_0$
$\texttt{m}_0$	\$
$\texttt{p}_0$	$\texttt{p}_1$
$\texttt{p}_1$	$\texttt{i}_1$
$\texttt{s}_0$	$\texttt{s}_2$
$\texttt{s}_1$	$\texttt{s}_3$
$\texttt{s}_2$	$\texttt{i}_2$
$\texttt{s}_3$	$\texttt{i}_3$

We look up the previous letter in the F column and take the corresponding letter from the L column.

Example

$$T = \texttt{mississippi\$}$$
$$\text{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
\$	$\texttt{i}_0$
$\texttt{i}_0$	$\texttt{p}_0$
$\texttt{i}_1$	$\texttt{s}_0$
$\texttt{i}_2$	$\texttt{s}_1$
$\texttt{i}_3$	$\texttt{m}_0$
$\texttt{m}_0$	\$
$\texttt{p}_0$	$\texttt{p}_1$
$\texttt{p}_1$	$\texttt{i}_1$
$\texttt{s}_0$	$\texttt{s}_2$
$\texttt{s}_1$	$\texttt{s}_3$
$\texttt{s}_2$	$\texttt{i}_2$
$\texttt{s}_3$	$\texttt{i}_3$

We look up the previous letter in the F column and take the corresponding letter from the L column.

Example

$$T = \texttt{mississippi\$}$$
$$\mathrm{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
\$	$\mathtt{i}_0$
$\mathtt{i}_0$	$\mathtt{p}_0$
$\mathtt{i}_1$	$\mathtt{s}_0$
$\mathtt{i}_2$	$\mathtt{s}_1$
$\mathtt{i}_3$	$\mathtt{m}_0$
$\mathtt{m}_0$	\$
$\mathtt{p}_0$	$\mathtt{p}_1$
$\mathtt{p}_1$	$\mathtt{i}_1$
$\mathtt{s}_0$	$\mathtt{s}_2$
$\mathtt{s}_1$	$\mathtt{s}_3$
$\mathtt{s}_2$	$\mathtt{i}_2$
$\mathtt{s}_3$	$\mathtt{i}_3$

We look up the previous letter in the F column and take the corresponding letter from the L column.

Example

$$T = \texttt{mississippi\$}$$
$$\mathrm{BWT}(T) = \texttt{ipssm\$pissii}$$

F	L
$\$$	$\texttt{i}_0$
$\texttt{i}_0$	$\texttt{p}_0$
$\texttt{i}_1$	$\texttt{s}_0$
$\texttt{i}_2$	$\texttt{s}_1$
$\texttt{i}_3$	$\texttt{m}_0$
$\texttt{m}_0$	$\$$
$\texttt{p}_0$	$\texttt{p}_1$
$\texttt{p}_1$	$\texttt{i}_1$
$\texttt{s}_0$	$\texttt{s}_2$
$\texttt{s}_1$	$\texttt{s}_3$
$\texttt{s}_2$	$\texttt{i}_2$
$\texttt{s}_3$	$\texttt{i}_3$

We end up with the string $\texttt{mississippi\$}$.

Today's lecture

1 Computation and compression

2 Reversal

3 Pattern matching

Searching with BWT

- Say we are searching our string T for occurrences of a substring P.
- Because the BWM is sorted, all rotations starting with an occurrence of P appear consecutively.
- Moreover, all occurrences of *substrings* of P occur consecutively as well.
- We use this to our advantage with a technique called backwards matching.

Example

T = ababbaba\$

BWT(T) = abb\$babaa

P = aba

	F		L
0	\$	ababbab	a_0
1	a_0	\$ababba	b_0
2	a_1	ba\$abab	b_1
3	a_2	babbaba	\$
4	a_3	bbaba\$a	b_2
5	b_0	a\$ababb	a_1
6	b_1	aba\$aba	b_3
7	b_2	abbaba\$	a_2
8	b_3	baba\$ab	a_3

We start by creating the ranked BWM, like we did for reversing the BWT.

Example

$$T = \texttt{ababbaba\$}$$
$$\mathrm{BWT}(T) = \texttt{abb\$babaa}$$

$$P = \texttt{aba}$$

	F		L
0	\$	ababbab	a_0
1	a_0	\$ababba	b_0
2	a_1	ba\$abab	b_1
3	a_2	babbaba	\$
4	a_3	bbaba\$a	b_2
5	b_0	a\$ababb	a_1
6	b_1	aba\$aba	b_3
7	b_2	abbaba\$	a_2
8	b_3	baba\$ab	a_3

We first look for rows starting with a. These are in the range $[1, 4]$.

Example

$$T = \texttt{ababbaba\$}$$
$$\text{BWT}(T) = \texttt{abb\$babaa}$$

$$P = \texttt{aba}$$

	F		L
0	\$	ababbab	a_0
1	a_0	\$ababba	b_0
2	a_1	ba\$abab	b_1
3	a_2	babbaba	\$
4	a_3	bbaba\$a	b_2
5	b_0	a\$ababb	a_1
6	b_1	aba\$aba	b_3
7	b_2	abbaba\$	a_2
8	b_3	baba\$ab	a_3

Next, we look for rows starting with ba. Within the range for a, these are the rows with a b in the L column, i.e. b_0 through b_2.

Example

$$T = \texttt{ababbaba\$}$$
$$\mathrm{BWT}(T) = \texttt{abb\$babaa}$$

$$P = \texttt{aba}$$

	F		L
0	\$	ababbab	a_0
1	a_0	\$ababba	b_0
2	a_1	ba\$abab	b_1
3	a_2	babbaba	\$
4	a_3	bbaba\$a	b_2
5	$\boxed{b_0}$	a\$ababb	a_1
6	$\boxed{b_1}$	aba\$aba	b_3
7	$\boxed{b_2}$	abbaba\$	a_2
8	b_3	baba\$ab	a_3

We find b_0 through b_2 in rows $[5, 7]$. Indeed, these start with ba.

Example

$T = \mathtt{ababbaba\$}$

$\mathrm{BWT}(T) = \mathtt{abb\$babaa}$

$P = \mathtt{aba}$

	F		L
0	\$	ababbab	a_0
1	a_0	\$ababba	b_0
2	a_1	ba\$abab	b_1
3	a_2	babbaba	\$
4	a_3	bbaba\$a	b_2
5	b_0	a\$ababb	a_1
6	b_1	aba\$aba	b_3
7	b_2	abbaba\$	a_2
8	b_3	baba\$ab	a_3

To find rows starting with aba, we look in the L column again to find a_1 and a_2.

Example

$T = \texttt{ababbaba\$}$

$\mathrm{BWT}(T) = \texttt{abb\$babaa}$

$P = \texttt{aba}$

	F		L
0	\$	ababbab	a_0
1	a_0	\$ababba	b_0
2	a_1	ba\$abab	b_1
3	a_2	babbaba	\$
4	a_3	bbaba\$a	b_2
5	b_0	a\$ababb	a_1
6	b_1	aba\$aba	b_3
7	b_2	abbaba\$	a_2
8	b_3	baba\$ab	a_3

This takes us to rows $[2, 3]$, concluding our search.

Pattern matching

- If at any point in the backward matching algorithm, we do not find the right letter in the L column, the pattern does not occur in T.
- Otherwise, the algorithm returns a range of rows in the BWM indicating the pattern's occurrences.
- If we want to find the actual positions in the original string, we need some extra steps
  - One way of doing this is to pick a set of "anchor points" in T, for which we map the corresponding BWM rows to the positions in the string.
  - Then, after finding an occurrence of P, we use LF mapping to find the previous anchor point to obtain the position in T.

rankAll

- A drawback is that, in each step, we are scanning a range of elements in L. This is $\mathcal{O}(n)$ (where $n = |T|$).
- We can make this $\mathcal{O}(1)$ by augmenting the ranks in the following way.
- Instead of storing the L-column ranks in a $n \times 1$ array, we store a $n \times |\Sigma|$ rankAll matrix, which stores for every letter the number of occurrences up to each row.

rankAll

$$T = \texttt{ababbaba\$}$$
$$\mathrm{BWT}(T) = \texttt{abb\$babaa}$$

	F		L	\$ a b
0	\$	ababbab	a	0 1 0
1	a_0	\$ababba	b	0 1 1
2	a_1	ba\$abab	b	0 1 2
3	a_2	babbaba	\$	1 1 2
4	a_3	bbaba\$a	b	1 1 3
5	b_0	a\$ababb	a	1 2 3
6	b_1	aba\$aba	b	1 2 4
7	b_2	abbaba\$	a	1 3 4
8	b_3	baba\$ab	a	1 4 4

In each row of rankAll, we store for each letter of the alphabet the number of times that letter occurs up to and including that row in L.

rankAll

$$T = \texttt{ababbaba\$}$$
$$\mathrm{BWT}(T) = \texttt{abb\$babaa}$$

	F	L	\$ a b
0	\$ ababbab	a	0 1 0
1	a_0 \$ababba	b	0 1 1
2	a_1 ba\$abab	b	0 1 2
3	a_2 babbaba	\$	1 1 2
4	a_3 bbaba\$a	b	1 1 3
5	b_0 a\$ababb	a	1 2 3
6	b_1 aba\$aba	b	1 2 4
7	b_2 abbaba\$	a	1 3 4
8	b_3 baba\$ab	a	1 4 4

In each row of rankAll, we store for each letter of the alphabet the number of times that letter occurs up to and including that row in L.

rankAll

$T =$ ababbaba\$

$\mathrm{BWT}(T) =$ abb\$babaa

$P =$ aba

	F		L	\$	a	b
0	\$	ababbab	a	0	1	0
1	a_0	\$ababba	b	0	1	1
2	a_1	ba\$abab	b	0	1	2
3	a_2	babbaba	\$	1	1	2
4	a_3	bbaba\$a	b	1	1	3
5	b_0	a\$ababb	a	1	2	3
6	b_1	aba\$aba	b	1	2	4
7	b_2	abbaba\$	a	1	3	4
8	b_3	baba\$ab	a	1	4	4

Say we have matched ba and are now trying to find occurrences of aba, as we did before.

Now we only have to look up the rankAll values for a at the start and end of our range.

rankAll

$$T = \texttt{ababbaba\$}$$
$$\mathrm{BWT}(T) = \texttt{abb\$babaa}$$

$$P = \texttt{aba}$$

	F		L	\$	a	b
0	\$	ababbab	a	0	1	0
1	a_0	\$ababba	b	0	1	1
2	a_1	ba\$abab	b	0	1	2
3	a_2	babbaba	\$	1	1	2
4	a_3	bbaba\$a	b	1	1	3
5	b_0	a\$ababb	a	1	2	3
6	b_1	aba\$aba	b	1	2	4
7	b_2	abbaba\$	a	1	3	4
8	b_3	baba\$ab	a	1	4	4

There is 1 a (a_0) *before* our range.
There are 3 as ($a_0 - a_2$) *before the end of* our range.
This tells us that the range contains a_1 and a_2 in the L column.

rankAll

$$T = \texttt{ababbaba\$}$$
$$\text{BWT}(T) = \texttt{abb\$babaa}$$

$$P = \texttt{aab}$$

	F		L	\$	a	b
0	\$	ababbab	a	0	1	0
1	a_0	\$ababba	b	0	1	1
2	a_1	ba\$abab	b	0	1	2
3	a_2	babbaba	\$	1	1	2
4	a_3	bbaba\$a	b	1	1	3
5	b_0	a\$ababb	a	1	2	3
6	b_1	aba\$aba	b	1	2	4
7	b_2	abbaba\$	a	1	3	4
8	b_3	baba\$ab	a	1	4	4

Now suppose we have matched ab, and want to extend this to aab.

rankAll

$$T = \texttt{ababbaba\$}$$
$$\mathrm{BWT}(T) = \texttt{abb\$babaa}$$

$P = \texttt{aab}$

	F	L	\$ a b
0	\$ ababbab	a	0 1 0
1	a_0 \$ababba	b	0 1 1
2	a_1 ba\$abab	b	0 1 2
3	a_2 babbaba	\$	1 1 2
4	a_3 bbaba\$a	b	1 1 3
5	b_0 a\$ababb	a	1 2 3
6	b_1 aba\$aba	b	1 2 4
7	b_2 abbaba\$	a	1 3 4
8	b_3 baba\$ab	a	1 4 4

The rank values are equal, which tells us that there is no match for aab.

Mapping F ranks

Without the optimization we are checking whether each letter matches our search letter in a for loop

- We can also optimize the lookups of ranks in the F column.
- Right now, looking up a single ranked letter takes $\mathcal{O}(n)$ time.
- We can reduce this to $\mathcal{O}(1)$ as well, by creating an extra array C that maps c_0 for every $c \in \Sigma$ to its corresponding row in the BWM.

Mapping F ranks

$$T = \texttt{ababbaba\$}$$

$$\mathrm{BWT}(T) = \texttt{abb\$babaa}$$

Σ	C
\$	0
a	1
b	5

	F	L	\$	a	b
0	\$	a	0	1	0
1	a	b	0	1	1
2	a	b	0	1	2
3	a	\$	1	1	2
4	a	b	1	1	3
5	b	a	1	2	3
6	b	b	1	2	4
7	b	a	1	3	4
8	b	a	1	4	4

Each entry of C points to the first occurrence of a letter in the F column of the BWM.

Mapping F ranks

$$T = \text{ababbaba\$}$$

$$\text{BWT}(T) = \text{abb\$babaa}$$

Σ	C
\$	0
a	1
b	5

	F	L	\$	a	b
0	\$	a	0	1	0
1	a	b	0	1	1
2	a	b	0	1	2
3	a	\$	1	1	2
4	a	b	1	1	3
5	b	a	1	2	3
6	b	b	1	2	4
7	b	a	1	3	4
8	b	a	1	4	4

+2

Say we want to find b_2 in the F column. Then we first go to row $C[\text{b}]$, and then move down two rows.

Mapping F ranks

$$T = \text{ababbaba\$}$$

$$\text{BWT}(T) = \text{abb\$babaa}$$

Σ	C
\$	0
a	1
b	5

	F	L	\$	a	b
0	\$	a	0	1	0
1	a	b	0	1	1
2	a	b	0	1	2
3	a	\$	1	1	2
4	a	b	1	1	3
5	b	a	1	2	3
6	b	b	1	2	4
7	b	a	1	3	4
8	b	a	1	4	4

In general, we can find letter c_j in row $C[c] + j$ for any $c \in \Sigma$.

Mapping F ranks

$$T = \texttt{ababbaba\$}$$

$$\mathrm{BWT}(T) = \texttt{abb\$babaa}$$

Σ	C
\$	0
a	1
b	5

	L	\$ a b
0	a	0 1 0
1	b	0 1 1
2	b	0 1 2
3	\$	1 1 2
4	b	1 1 3
5	a	1 2 3
6	b	1 2 4
7	a	1 3 4
8	a	1 4 4

We no longer need to explicitly store the F column, as the C table contains the same information.

Mapping F ranks

$$T = \mathtt{ababbaba\$}$$
$$\mathrm{BWT}(T) = \mathtt{abb\$babaa}$$

Σ	C
\$	0
a	1
b	5

	L	\$	a	b
0	a	0	1	0
1	b	0	1	1
2	b	0	1	2
3	\$	1	1	2
4	b	1	1	3
5	a	1	2	3
6	b	1	2	4
7	a	1	3	4
8	a	1	4	4

To compute C without the F column:
Each entry $C[c]$ is equal to the total number of occurrences of letters “smaller” than c in BWT(T).

Complexity of reversal and pattern matching

- If we implement rankAll and the C array, we can:
  - reverse the BWT in $\mathcal{O}(n)$ time; and
  - count the number of occurrences of a pattern P in $\mathcal{O}(m)$ time, where $m = |P|$.
- Instead of the rankAll matrix, we can use a data structure called wavelet tree to the same end [2].
  - This reduces its space usage from $n \cdot |\Sigma|$ to $\mathcal{O}(n \log |\Sigma|)$.
  - Looking up ranks now takes $\mathcal{O}(\log |\Sigma|)$ time, so pattern matching takes $\mathcal{O}(m \log |\Sigma|)$ time.

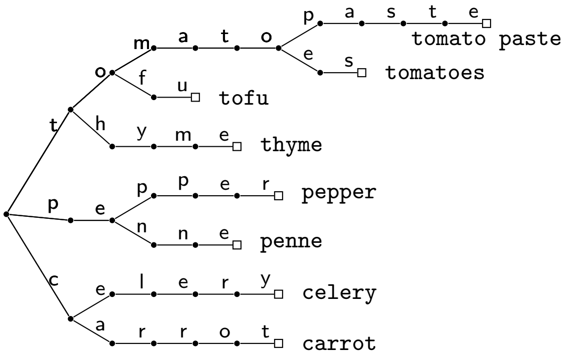
Faster construction of the BWT

Rotate the strings then sort

- Constructing the BWT by sorting rotations takes at best $\mathcal{O}(n^2)$ time.
  - This is too slow for large texts or DNA sequences.
- There are algorithms that construct the BWT directly in $\mathcal{O}(n)$ time [3].
- One indirect method is by first constructing the suffix array.
- We will take a very brief look at the suffix tree and suffix array now, to see their connection to the BWT.

Tries

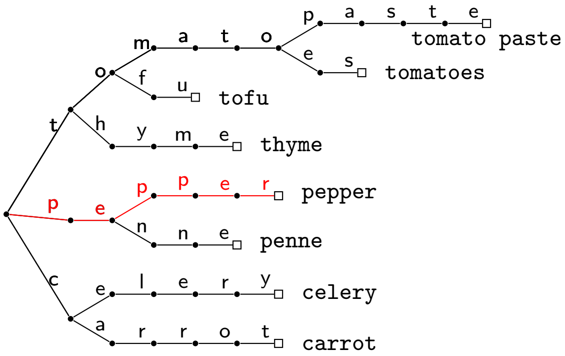
The trie of a set of strings is a tree with labeled edges, in which each root-to-leaf path spells one of the encoded strings.



{carrot, celery, penne, pepper, thyme, tofu, tomatoes, tomato paste}

Tries

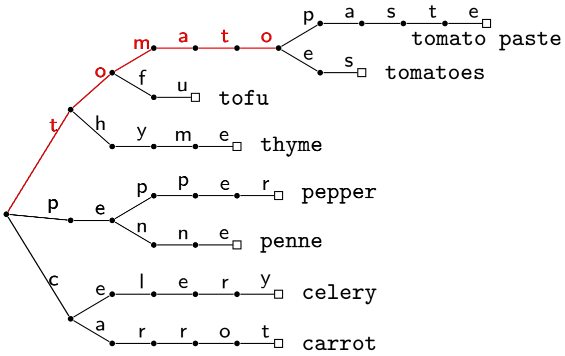
The trie of a set of strings is a tree with labeled edges, in which each root-to-leaf path spells one of the encoded strings.



$\{$carrot, celery, penne, pepper, thyme, tofu, tomatoes, tomato paste$\}$

Tries

The trie of a set of strings is a tree with labeled edges, in which each root-to-leaf path spells one of the encoded strings.

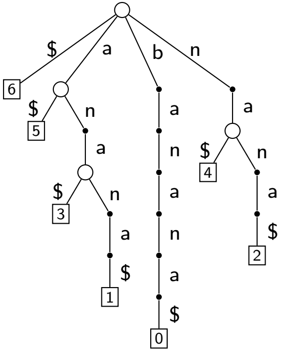


{carrot, celery, penne, pepper, thyme, tofu, tomatoes, tomato paste}

Suffix trees

Given a string T, the trie of all of T's suffixes is called suffix tree.

$T =$ `banana$` (0)
`anana$` (1)
`nana$` (2)
`ana$` (3)
`na$` (4)
`a$` (5)
`$` (6)

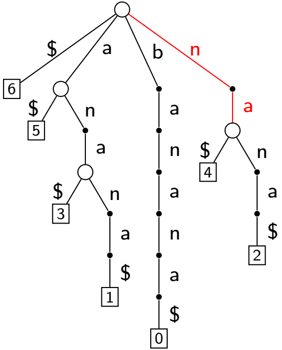


We label the leaves of the suffix trees with the starting positions of the corresponding suffixes. For each internal node we sort its children alphabetically.

Suffix trees

Given a string T, the trie of all of T's suffixes is called suffix tree.

$T =$ `banana$` (0)
`anana$` (1)
`nana$` (2)
`ana$` (3)
`na$` (4)
`a$` (5)
`$` (6)



The suffix tree can also be used for pattern matching: e.g. if we follow the pattern `na`, we find suffixes 2 and 4.

Suffix array

A related data structure is the suffix array, which is obtained by sorting all of T's suffixes.

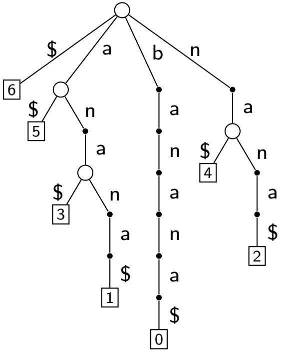
$$
\begin{array}{rl}
T = \texttt{banana\$} & (0) \\
\texttt{anana\$} & (1) \\
\texttt{nana\$} & (2) \\
\texttt{ana\$} & (3) \\
\texttt{na\$} & (4) \\
\texttt{a\$} & (5) \\
\texttt{\$} & (6)
\end{array}
\quad \longrightarrow \quad
\begin{array}{ll}
\texttt{\$} & (6) \\
\texttt{a\$} & (5) \\
\texttt{ana\$} & (3) \\
\texttt{anana\$} & (1) \\
\texttt{banana\$} & (0) \\
\texttt{na\$} & (4) \\
\texttt{nana\$} & (2)
\end{array}
$$

We store the starting positions of the sorted suffixes in an array $\mathrm{SA}(\texttt{banana\$}) = (6, 5, 3, 1, 0, 4, 2)$.

Suffix tree and suffix array

The suffix array can also be found by reading the leaves of the suffix tree left to right:

`$`	(6)
`a$`	(5)
`ana$`	(3)
`anana$`	(1)
`banana$`	(0)
`na$`	(4)
`nana$`	(2)



Both the suffix array and suffix tree are common and useful tools in string algorithms.

BWT via the suffix array

The suffix array is also closely related to the BWT. This is clear when written side by side with the BWM:

BWM	SA
$banana	$
a$banan	a$
ana$ban	ana$
anana$b	anana$
banana$	banana$
na$bana	na$
nana$ba	nana$

Kärkkäinen-Sanders Algorithm

Theorem (Kärkkäinen & Sanders, 2003 [4])

For a string of length n, the suffix array can be computed in $\mathcal{O}(n)$ time.

Conclusion

- The Burrows-Wheeler Transform permutes the letters of a string T into another string $\mathrm{BWT}(T)$.
- Its main utility lies in the fact that the BWT of structured data often contains long runs, which can be compressed using run-length encoding.
- Using some auxiliary data structures, we can:
  - compute and reverse the BWT in linear time; and
  - use the BWT to count the occurrences of a length-m pattern in $\mathcal{O}(m)$ time.
- There exist $\mathcal{O}(n)$-time algorithms to compute the BWT. One method achieves this by computing the suffix array first.

It can also be done in O(n) time without computing the suffix array, for example, BWT-IS algorithm

Bibliography I

M. Burrows and D. J. Wheeler.
A block-sorting lossless data compression algorithm.
Technical Report SRC-RR-124, Standord Univeristy, 1994.

Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter.
High-order entropy-compressed text indexes.
In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*, pages 841–850. ACM/SIAM, 2003.

Juha Kärkkäinen.
Fast BWT in small space by blockwise suffix sorting.
Theor. Comput. Sci., 387(3):249–257, November 2007.

Bibliography II

Juha Kärkkäinen and Peter Sanders.
Simple linear work suffix array construction.
In *Proceedings of the 30th International Conference on Automata, Languages and Programming*, ICALP'03, pages 943–955, Berlin, Heidelberg, 2003. Springer-Verlag.

Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg.
Ultrafast and memory-efficient alignment of short DNA sequences to the human genome.
Genome biology, 10(3):R25+, 2009.

Heng Li and Richard Durbin.
Fast and accurate short read alignment with Burrows-Wheeler transform.
Bioinformatics, 25(14):1754–1760, 2009.