A_stree reflection

**Time**

**Learning Objectives met**

**Lecture Helpfulness**

**Dynamic iterator was well-liked**
A_sarray due today!

Remember to return all matching strings!
Exact pattern matching \textit{w/ indexing}

There are many data structures built on \textit{suffixes}

Before break we looked at these

- Suffix Trie
- Suffix Tree
- Suffix Array
- FM Index
**Exact pattern matching w/ indexing**

<table>
<thead>
<tr>
<th></th>
<th>Suffix tree</th>
<th>Suffix array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time: Does P occur?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time: Report $k$ locations of P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Space</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$m = |T|$, $n = |P|$, $k = \#$ occurrences of $P$ in $T$
Suffix tree vs suffix array: size

The suffix array has a smaller constant factor than the tree

- Suffix tree: ~16 bytes per character
- Suffix array: ~4 bytes per character
- Raw text: 2 bits per character
Exact pattern matching *w*/* indexing

There are many data structures built on **suffixes**

The FM index is a compressed self-index (smaller* than original text)!
Exact pattern matching \textit{w/ indexing}

The basis of the FM index is a \textit{transformation}

\begin{align*}
  \text{B A N A N A } & \text{ $} \\
  \downarrow \\
  \text{A N N B } & \text{ $ A A }
\end{align*}
Burrows-Wheeler Transform

*Reversible permutation* of the characters of a string

<table>
<thead>
<tr>
<th>T</th>
<th>B W T(T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B A N A N A $</td>
<td>A N N B $ A A</td>
</tr>
</tbody>
</table>

1) How to encode?

2) How to decode?

3) How is it useful for search?
Burrows-Wheeler Transform

*Reversible permutation* of the characters of a string

$abaabab$  $T$  $???$

All rotations

Text rotations

A string is a ‘rotation’ of another string if it can be reached by wrap-around shifting the characters

![Diagram showing text rotations](image-url)

(after this they repeat)
Text Rotations

A string is a ‘rotation’ of another string if it can be reached by wrap-around shifting the characters

Which of these are rotations of ‘ABCD’?

A) BCDA  
B) BACD
C) DCAB  
D) CDAB
Burrows-Wheeler Transform

*Reversible permutation* of the characters of a string

(after this they repeat)

Burrows-Wheeler Transform

Reversible permutation of the characters of a string

\[ \text{a b a a b a $} \]

\[
\begin{align*}
\text{a b a a b a $} & \\
\text{$ a b a a b a} & \\
\text{a $ a b a a b} & \\
\text{b a $ a b a a} & \\
\text{a b a $ a b a} & \\
\text{a a b a $ a b} & \\
\text{b a a b a $ a} & \\
\end{align*}
\]

All rotations

Burrows-Wheeler Transform

*Reversible permutation* of the characters of a string

$$a \ b \ a \ a \ b \ a \ a \ b$$

$T$

All rotations

Sort

Burrows-Wheeler Matrix

$\begin{align*}
& a \ b \ a \ a \ b \\
& a \ a \ b \ a \ a \ b \\
& a \ b \ a \ a \ b \ a \ b \\
& a \ b \ a \ a \ b \ a \ b \\
& b \ a \ a \ b \ a \ a \ b \\
& b \ a \ a \ b \ a \ a \ b \\
& b \ a \ a \ b \ a \ a \ b
\end{align*}$

BWT($T$)

Last column

Burrows-Wheeler Transform

(1) Build all rotations
(2) Sort all rotations
(3) Take last column

\[ T = \text{c a r } \$ \]
Burrows-Wheeler Transform

(1) Build all rotations
(2) Sort all rotations
(3) Take last column

$T = \text{c a r }$ $\rightarrow$ $\text{$c a r$}$ $\rightarrow$ $\text{r c $a$}$
Assignment 8: a_bwt

Learning Objective:

Implement the Burrows-Wheeler Transform on text

Reverse the Burrows-Wheeler Transform to reproduce text

Consider: How can the BWT be stored smaller than the original text?
Burrows-Wheeler Transform

How to reverse the BWT?

All rotations

Sort

Burrows-Wheeler Matrix

$ a \ b \ a \ a \ b \ a \ a \ b 
\ a \ a \ b \ a \ a \ b 
\ a \ a \ b \ a \ a \ b 
\ a \ b \ a \ a \ b \ a 
\ a \ b \ a \ a \ b \ a 
\ b \ a \ a \ b \ a \ a 
\ b \ a \ a \ b \ a \ a $

BWT(T)

Last column
Burrows-Wheeler Transform

\[ \text{BWT}(T) = r \ c \ $ \ a \quad T = c \ a \ r \ $ \]
Burrows-Wheeler Transform

BWT(T) = r c $ a \quad T = c a r$

1) Prepend the BWT as a column
2) Sort the full matrix as rows
3) Repeat 1 and 2 until full matrix
4) Pick the row ending in ‘$’
Burrows-Wheeler Transform

\[ \text{BWT}(T) = r \ c \ \$ \ a \quad T = c \ a \ r \ \$ \]
Burrows-Wheeler Transform

\[ \text{BWT}(T) = r \ c \ $ \ a \ \ \ \ \ T = c \ a \ r \ $ \]

$ \ c \ a \ r \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ $ \ c $

a \ r \ $ \ c \ a \ r \ $ \ c \ a $

r \ $ \ c \ a \ r \ $ \ r \ $
Burrows-Wheeler Transform

\[ BWT(T) = r\ c\ \$\ a \quad T = c\ a\ r\ \$ \]

\[
\begin{array}{cccc}
\$, & c & a & r \\
a & r & \$ & c \\
c & a & r & \$
\end{array}
\quad
\begin{array}{cccc}
\$, & c & a \\
a & r & \$ \\
c & a & r \\
r & \$ & c
\end{array}
\]
Burrows-Wheeler Transform

What is the right context of apple? le$ap

A letter always has the same right context.

$apple
apple$
el$apple
le$ap
ple$ap
ple$ap
ple$ap
Burrows-Wheeler Transform: T-ranking

To continue, we have to be able to uniquely identify each character in our text.

Give each character in $T$ a rank, equal to the number of times the character occurred previously in $T$. Call this the $T$-ranking.

\[ a \ b \ a \ a \ b \ a \ S \]

Ranks aren’t explicitly stored; they are just for illustration.
Burrows-Wheeler Transform

BWM with T-ranking:

\[
\begin{array}{cccc}
F & & & \\
\$ & a_0 & b & a_1 & a_2 & b & a_3 \\
a_3 & $ & a_0 & b & a_1 & a_2 & b \\
a_1 & a_2 & b & a_3 & $ & a_0 & b \\
a_2 & b & a_3 & $ & a_0 & b & a_1 \\
a_0 & b & a_1 & a_2 & b & a_3 & $ \\
b & a_3 & $ & a_0 & b & a_1 & a_2 \\
b & a_1 & a_2 & b & a_3 & $ & a_0 \\
\end{array}
\]

Look at first and last columns, called \( F \) and \( L \) (and look at just the \( a \)s)

\( a \)s occur in the same order in \( F \) and \( L \). As we look down columns, in both cases we see: \( a_3, a_1, a_2, a_0 \)
Burrows-Wheeler Transform

BWM with T-ranking:

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$ a_0</td>
<td>b a_1</td>
</tr>
<tr>
<td>b</td>
<td>a_2</td>
<td>b a_3</td>
</tr>
<tr>
<td>$</td>
<td>a_3</td>
<td>$ a_0</td>
</tr>
<tr>
<td>a</td>
<td>a_0</td>
<td>b a_1</td>
</tr>
<tr>
<td>b</td>
<td>a_2</td>
<td>b a_3</td>
</tr>
<tr>
<td>a</td>
<td>a_0</td>
<td>b a_1</td>
</tr>
<tr>
<td>b</td>
<td>a_3</td>
<td>$ a_0</td>
</tr>
<tr>
<td>b</td>
<td>a_1</td>
<td>b a_3</td>
</tr>
<tr>
<td>a</td>
<td>$ a_0</td>
<td>b a_1</td>
</tr>
<tr>
<td>b</td>
<td>a_2</td>
<td>b a_3</td>
</tr>
</tbody>
</table>

Same with bs:  b_1, b_0
Burrows-Wheeler Transform: LF Mapping

BWM with T-ranking:

<table>
<thead>
<tr>
<th></th>
<th>$</th>
<th>a_0</th>
<th>b</th>
<th>a_1</th>
<th>a_2</th>
<th>b</th>
<th>a_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_3</td>
<td></td>
<td>$</td>
<td>a_0</td>
<td>b</td>
<td>a_1</td>
<td>a_2</td>
<td>b</td>
</tr>
<tr>
<td>a_1</td>
<td>a_2</td>
<td>b</td>
<td>a_0</td>
<td>$</td>
<td>a_0</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a_2</td>
<td>b</td>
<td>a_3</td>
<td>$</td>
<td>a_0</td>
<td>b</td>
<td>a_1</td>
<td></td>
</tr>
<tr>
<td>a_0</td>
<td>b</td>
<td>a_1</td>
<td>a_2</td>
<td>b</td>
<td>a_3</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>a_3</td>
<td>$</td>
<td>a_0</td>
<td>b</td>
<td>a_1</td>
<td>a_2</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>a_1</td>
<td>a_2</td>
<td>b</td>
<td>a_3</td>
<td>$</td>
<td>a_0</td>
<td></td>
</tr>
</tbody>
</table>

LF Mapping: The $i^{th}$ occurrence of a character $c$ in $L$ and the $i^{th}$ occurrence of $c$ in $F$ correspond to the same occurrence in $T$ (i.e. have same rank)

This works because all our strings are rotations!
Burrows-Wheeler Transform: LF Mapping

Why does this work?

These characters have the same right contexts!

These characters are the same character!
Burrows-Wheeler Transform: LF Mapping

Why does this work?

Why are these $a$s in this order relative to each other?

Occurrences of $c$ in $F$ are sorted by right-context. Same for $L$!

Any ranking we give to characters in $T$ will match in $F$ and $L$
Burrows-Wheeler Transform: LF Mapping

LF Mapping can be used to recover our original text too!

Given BWT = a₃ b₁ b₀ a₁ $ a₂ a₀

What is L?

What is F?
Burrows-Wheeler Transform: LF Mapping

LF Mapping can be used to recover our original text too!

Start in first row. $F$ must have $.$
$L$ contains character just prior to $\$: $a_3$

Jump to row beginning with $a_0$.
$L$ contains character just prior to $a_0$: $b_0$.

Repeat for $b_0$, get $a_2$
Repeat for $a_2$, get $a_1$
Repeat for $a_1$, get $b_1$
Repeat for $b_1$, get $a_3$
Repeat for $a_3$, get $\$ (done)
**Burrows-Wheeler Transform: LF Mapping**

Another way to visualize:

<p>| | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>L</td>
<td>F</td>
<td>L</td>
<td>F</td>
<td>L</td>
<td>F</td>
<td>L</td>
<td>F</td>
<td>L</td>
<td>F</td>
<td>L</td>
<td>F</td>
<td>L</td>
<td>F</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
</tr>
</tbody>
</table>

**T:** \[a_0 \ b_0 \ a_1 \ a_2 \ b_1 \ a_3 \ $]
Assignment 8: a_bwt

Learning Objective:

Implement the Burrows-Wheeler Transform on text

Reverse the Burrows-Wheeler Transform to reproduce text

Consider: You can use either LF mapping or prepend-sort to reverse. Which do you think would be easier to implement (or more efficient)?
### Burrows-Wheeler Transform: A better ranking

**Any ranking** we give to characters in \( T \) will match in \( F \) and \( L \)

<table>
<thead>
<tr>
<th>T-Rank: Order by T</th>
<th>F-Rank: Order by F</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F )</td>
<td>( L )</td>
</tr>
<tr>
<td>$</td>
<td>( a_3 )</td>
</tr>
<tr>
<td>( a_3 )</td>
<td>( b_1 )</td>
</tr>
<tr>
<td>( a_1 )</td>
<td>( b_0 )</td>
</tr>
<tr>
<td>( a_2 )</td>
<td>( a_1 )</td>
</tr>
<tr>
<td>( a_0 )</td>
<td>( $ )</td>
</tr>
<tr>
<td>( b_1 )</td>
<td>( a_2 )</td>
</tr>
<tr>
<td>( b_0 )</td>
<td>( a_0 )</td>
</tr>
</tbody>
</table>

\( F \)-rank is easy to store!
Burrows-Wheeler Transform: A better ranking

$T = \ a \ b \ b \ c \ c \ d \ \$\$

What is the BWM index for my first instance of C? ($C_0$) [0-base for answer]

<table>
<thead>
<tr>
<th>F</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>a b b c c d</td>
</tr>
<tr>
<td>a</td>
<td>b b c c d $</td>
</tr>
<tr>
<td>b</td>
<td>b c c d $ a</td>
</tr>
<tr>
<td>b</td>
<td>c c d $ a b</td>
</tr>
<tr>
<td>c</td>
<td>c d $ a b b</td>
</tr>
<tr>
<td>c</td>
<td>d $ a b b c</td>
</tr>
<tr>
<td>d</td>
<td>$ a b b c c</td>
</tr>
</tbody>
</table>
Burrows-Wheeler Transform: A better ranking

Say $T$ has 300 $A$s, 400 $C$s, 250 $G$s and 700 $T$s and $\$ < A < C < G < T$

What is the BWM index for my 100th instance of $G$? ($G^{99}$) [0-base for answer]

- Skip row starting with $\$$(1 row)
- Skip rows starting with $A$ (300 rows)
- Skip rows starting with $C$ (400 rows)
- Skip first 99 rows starting with $G$ (99 rows)

**Answer:** skip 800 rows -> **index 800 contains my 100th G**

With a little preprocessing we can skip 701 rows!
FM Index

An index combining the BWT with a few small auxiliary data structures

Core of index is **first (F)** and **last (L) rows** from BWM:

$L$ is the same size as $T$

$F$ can be represented as array of $|\Sigma|$ integers (or not stored at all!)

We’re discarding $T$ — we can recover it from $L$!
FM Index: Querying

Can we query like the suffix array?

We don’t have these columns, and we don’t have T. Binary search not possible.
FM Index: Querying

The BWM is a lot like the suffix array — maybe we can query the same way?

$ a b a a b a$
$a \ a b a a b$
$a a b a \ a b$
$a b a \ a b a$
$a b a a b a$
$b a \ a b a a$
$b a a b a \ a$

BWM(T)

6 $  
5 a $  
2 a a b a $  
3 a b a $  
0 a b a a b a $  
4 b a $  
1 b a a b a $  

SA(T)
FM Index: Querying

The BWM is a lot like the suffix array — maybe we can query the same way?

We don’t have these columns, and we don’t have T.
FM Index: Querying

Look for range of rows of BWM(T) with $P$ as prefix

Start with shortest suffix, then match successively longer suffixes

$$P = aba$$

$F$  $L$

$\$$  aba  a  b  a  a  b  a_0$

$a_0$  $\$$  a  b  a  a  b$

$a_1$  aba  $\$$  a  b$

$a_2$  ba  $\$$  a  b  a  a_1$

$a_3$  ba  ba  b  a  $\$$

b  a  $\$$  a  b  a  a_2$

b  a  a  b  a  $\$$  a_3$

Easy to find all the rows beginning with $a$
FM Index: Querying

We have rows beginning with $a$, now we want rows beginning with $ba$

\[ P = aba \]

Note: We still aren’t storing the characters in grey, we just know they exist.
FM Index: Querying

We have rows beginning with \textbf{ba}, now we seek rows beginning with \textbf{aba}

\[ P = \textbf{aba} \]

\[
\begin{array}{c|c}
F & L \\
\hline
$ & a b a a b a_0 \\
a_0 & $ a b a a b \\
a_1 & a b a $ a b \\
a_2 & b a $ a b a_1 \\
a_3 & b a a b a $ \\
\hline
b a & $ a b a a_2 \\
b & a a b a $ a_3 \\
\end{array}
\]

Use LF Mapping

\[ a_2, a_3 \text{ occur just to left.} \]

\[ P = \textbf{aba} \]

\[
\begin{array}{c|c}
F & L \\
\hline
$ & a b a a b a_0 \\
a_0 & $ a b a a b \\
a_1 & a b a $ a b \\
a_2 & b a $ a b a_1 \\
a_3 & b a a b a $ \\
\hline
b & a $ a b a_2 \\
b & a a b a $ a_3 \\
\end{array}
\]

Now we have the rows with prefix \textbf{aba}
FM Index: Querying

When \( P \) does not occur in \( T \), we eventually fail to find next character in \( L \):

\[
P = \text{bba}
\]

\[
\begin{array}{c|c}
F & L \\
\hline
\$ & a \ b \ a \ a \ b \ a_0 \\
a_0 & $ a \ b \ a \ a \ b \\
a_1 & a \ b \ a \ $ \ a \ b \\
a_2 & b \ a \ $ \ a \ b \ a_1 \\
a_3 & b \ a \ a \ b \ a \ $ \\
\end{array}
\]

Rows with \( \text{ba} \) prefix

\[
\text{b a $ a b a a}_2 \\
\text{b a a b a $ a}_3 \\
\]

← No bs!
Problem 1: If we scan characters in the last column, that can be slow, $O(m)$
**FM Index: Querying**

**Problem 2:** We don’t immediately know *where* the matches are in T...

\[ P = \text{aba} \]

Got the same range, \([3, 5)\), we would have got from suffix array

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>F</strong></td>
<td><strong>L</strong></td>
</tr>
<tr>
<td>$</td>
<td>a b a a b a_0</td>
</tr>
<tr>
<td>a_0</td>
<td>$ a b a a b</td>
</tr>
<tr>
<td>a_1</td>
<td>a b a a b</td>
</tr>
<tr>
<td>a_2</td>
<td>b a $ a b a_1</td>
</tr>
<tr>
<td>a_3</td>
<td>b a a b a</td>
</tr>
</tbody>
</table>

Where are the values?

Where are the values?

[3, 5)
Bonus Slides
Burrows-Wheeler Transform

*Reversible permutation* of the characters of a string

\[
\begin{align*}
T & \quad \text{BWT}(T) \\
B A N A N A $ & \quad \longleftrightarrow \quad A N N B $ A A
\end{align*}
\]

1) How to encode?

2) How to decode?

3) **How is it useful for compression?**

4) How is it useful for search?
Burrows-Wheeler Transform

Tomorrow_and_tomorrow_and_tomorrow

It was the best of times, it was the worst of times

“bzip”: compression w/ a BWT to better organize text
Burrows-Wheeler Transform

orrow_and_tomorrow_and_tomorrow$tom 
ow$tomorrow_and_tomorrow_and_tomorr 
ow_and_tomorrow$tomorrow_and_tomorr 
ow_and_tomorrow_and_tomorrow$tomorr 
ow$tomorrow_and_tomorrow_and_tomor 
ow_and_tomorrow$tomorrow_and_tomor 
ow_and_tomorrow_and_tomorrow$tomorr 
row$tomorrow_and_tomorrow_and_tomor 
ow_and_tomorrow$tomorrow_and_tomor 
ow_and_tomorrow_and_tomorrow$tomorr 
row$tomorrow_and_tomorrow_and_tomo

Ordered by the context to the right of each character
In English (and most languages),
the next character in a word is
not independent of the previous.

In general, if text structured
BWT(T) more compressible

![Figure 1: Example of sorted rotations. Twenty consecutive rotations from the
sorted list of rotations of a version of this paper are shown, together with the final
character of each rotation.](image)

Burrows-Wheeler Transform

Let's compare the SA with the BWT...

\[ T = \text{a b a a b a $} \]

| 6 | $ a b a a b a |
| 5 | a $ a b a a b |
| 2 | a a b a $ a b |
| 3 | a b a $ a b a |
| 0 | a b a a b a $ |
| 4 | b a $ a b a a |
| 1 | b a a b a $ a |

SA(T)  BWM(T)

Suffix Array is O(m)
Burrows-Wheeler Transform

Let's compare the SA with the BWT...

\[ T = a \ b \ a \ a \ b \ a \$ \]

\[
\begin{array}{cccccc}
6 & 5 & 2 & 3 & 0 & 4 \\
\end{array}
\]

\[
\begin{array}{cc}
\text{SA}(T) & \text{BWT}(T) \\
\end{array}
\]

Suffix Array is O(m)  BWT is O(m)

The BWT has a better constant factor!
Burrows-Wheeler Transform

BWM is related to the suffix array

<table>
<thead>
<tr>
<th>BWM(T)</th>
<th>SA(T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ a b a a b a$</td>
<td>6 $</td>
</tr>
<tr>
<td>a $ a b a a b</td>
<td>5 a $</td>
</tr>
<tr>
<td>a a b a $ a b</td>
<td>2 a a b a $</td>
</tr>
<tr>
<td>a b a $ a b a</td>
<td>3 a b a $</td>
</tr>
<tr>
<td>a b a a b a $</td>
<td>0 a b a a b a $</td>
</tr>
<tr>
<td>b a $ a b a a</td>
<td>4 b a $</td>
</tr>
<tr>
<td>b a a b a $ a</td>
<td>1 b a a b a $</td>
</tr>
</tbody>
</table>

Same order whether rows are rotations or suffixes
In fact, this gives us a new definition / way to construct BWT(T):

\[
BWT[i] = \begin{cases} 
T[SA[i] - 1] & \text{if } SA[i] > 0 \\
\$ & \text{if } SA[i] = 0 
\end{cases}
\]

“BWT = characters just to the left of the suffixes in the suffix array”
Burrows-Wheeler Transform

In fact, this gives us a new definition / way to construct BWT(T):

\[
BWT[i] = \begin{cases} 
  T[SA[i] - 1] & \text{if } SA[i] > 0 \\
  $ & \text{if } SA[i] = 0
\end{cases}
\]

“BWT = characters just to the left of the suffixes in the suffix array”