String Algorithms and Data Structures
Burrows-Wheeler Transform

CS 199-225
Brad Solomon

October 24, 2022

Department of Computer Science
Informal Early Feedback

The instructor is well-prepared for each class / recording

13 responses

- Strongly disagree: 7.7%
- Disagree: 46.2%
- Neutral: 46.2%
- Agree: 7.7%
- Strongly agree: 0%
Informal Early Feedback

I feel that I can actively participate in lecture
13 responses

23.1%

76.9%

I feel that I can actively participate in class in general
13 responses

15.4%

76.9%

7.7%
Informal Early Feedback

I receive helpful and complete answers to my questions

During lecture
- Strongly agree: 30.8%
- Agree: 61.5%
- Neutral: 7.7%

Outside lecture
- Strongly agree: 38.5%
- Agree: 53.8%
- Neutral: 7.7%
Informal Early Feedback

Lecture helpfulness

- 46.2% rate it 5 - Very helpful
- 7.7% rate it 4

Assignment helpfulness

- 61.5% rate it 5 - Very helpful
- 7.7% rate it 4
- 30.8% rate it 3
Informal Early Feedback

The discord is pretty useful, as the instructor often responds to answer questions.

It's hard to decide between the lectures and assignments. Both have been instrumental.

recorded lectures / slides [are the most helpful]

Getting some of the hidden test cases or charComps test cases is very difficult. I would like it more if these test cases were given or revealed, though I understand if this isn't possible.

Maybe spend more time on big O analysis. It's really confusing sometimes.

I wish we could move back to in-person lectures.
Exact pattern matching with indexing

There are many data structures built on *suffixes*

We have now seen both of these data structures.
# Exact pattern matching with 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 with indexing

There are many data structures built on suffixes.

The FM index is a compressed self-index (smaller* than original text)!

Reduced size

*Compressed compared to the original text size.
Exact pattern matching with indexing

The basis of the FM index is a transformation

B A N A N A $

A N N B $ A A
Burrows-Wheeler Transform

*Reversible permutation* of the characters of a string

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

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

\[
a b a a b a \$
\]

All rotations

Text rotations

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

\[ \text{a b c d e f } \]
\[ \text{b c d e f } \]
\[ \text{c d e f } \]
\[ \text{d e f } \]
\[ \text{e f } \]
\[ \text{f } \]
\[ \text{f a b c d e } \]
\[ \text{f a b c d e } \]
\[ \text{f a b c d e } \]
\[ \text{f a b c d e } \]
\[ \text{f a b c d e } \]
\[ \text{f a b c d e f } \]

(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

A) BCDA  B) BACD
C) DCAB  D) CDAB

Which of these are rotations of ‘ABCD’?
Burrows-Wheeler Transform

*Reversible permutation* of the characters of a string

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

(\text{after this they repeat})

Burrows-Wheeler Transform

*Reversible permutation* of the characters of a string

\[ a \ b \ a \ a \ b \ a \ \$ \]

\[ T \]

All rotations:

- \[ a \ b \ a \ a \ b \ a \ \$ \]
- \[ \$ \ a \ b \ a \ a \ b \ a \]
- \[ a \ \$ \ a \ b \ a \ a \ b \]
- \[ b \ a \ \$ \ a \ b \ a \ a \]
- \[ a \ b \ a \ \$ \ a \ b \ a \]
- \[ a \ a \ b \ a \ \$ \ a \ b \]
- \[ b \ a \ a \ b \ a \ \$ \ a \]

**Burrows-Wheeler Transform**

*Reversible permutation* of the characters of a string

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

---

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 } \$

All rotations

Sort

$\begin{align*}
\text{\$ c a r} \\
\text{a r \$ c} \\
\text{c a r \$} \\
\text{r \$ c a} \\
\text{r c \$ a}
\end{align*}$
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?
How to reverse the BWT?

Burrows-Wheeler Matrix

Sort

All rotations

BWT(T)

Last column

Burrows-Wheeler Transform
Burrows-Wheeler Transform

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

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

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

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

The Burrows-Wheeler Transform (BWT) is a data transformation that rearranges the characters of a given string in a way that makes certain types of compression algorithms more efficient. The transformation is reversible, meaning that the original string can be recovered from its transformed form.

Given a string $T = \text{car}$, its BWT is calculated as follows:

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

The transformed string is then arranged in lexicographical order, as shown:

<table>
<thead>
<tr>
<th>Original</th>
<th>BWT</th>
<th>Original</th>
<th>BWT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r c $ a</td>
<td>T</td>
<td>c a r $</td>
<td>BWT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a r $ c</td>
<td></td>
<td>a r $</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c a r $</td>
<td></td>
<td>c a r</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r $ c a</td>
<td></td>
<td>r $ c</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ c a r</td>
<td></td>
<td>$ c a</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Burrows-Wheeler Transform

What is the right context of \texttt{apple} ?

A letter always has the same right context.
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 $\#$ times the character occurred previously in $T$. Call this the $T$-ranking.

\[ a \quad b \quad a \quad a \quad b \quad a \quad a \quad $ \]

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

BWM with T-ranking:

\[
\begin{array}{cccccc}
F & L \\
\$ & a_0 & b_0 & a_1 & a_2 & b_1 & a_3 \\
a_3 & $ & a_0 & b_0 & a_1 & a_2 & b_1 \\
a_1 & a_2 & b_1 & a_3 & $ & a_0 & b_0 \\
a_2 & b_1 & a_3 & $ & a_0 & b_0 & a_1 \\
a_0 & b_0 & a_1 & a_2 & b_1 & a_3 & $ \\
b_1 & a_3 & $ & a_0 & b_0 & a_1 & a_2 \\
b_0 & a_1 & a_2 & b_1 & a_3 & $ & a_0 \\
\end{array}
\]

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:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>a₀</td>
<td>b₀</td>
<td>a₁</td>
</tr>
<tr>
<td>a₃</td>
<td>$</td>
<td>a₀</td>
<td>b₀</td>
</tr>
<tr>
<td>a₁</td>
<td>a₂</td>
<td>b₁</td>
<td>a₃</td>
</tr>
<tr>
<td>a₂</td>
<td>b₁</td>
<td>a₃</td>
<td>$</td>
</tr>
<tr>
<td>a₀</td>
<td>b₀</td>
<td>a₁</td>
<td>a₂</td>
</tr>
<tr>
<td>b₁</td>
<td>a₃</td>
<td>$</td>
<td>a₀</td>
</tr>
<tr>
<td>b₀</td>
<td>a₁</td>
<td>a₂</td>
<td>b₁</td>
</tr>
</tbody>
</table>

Same with bs: b₁, b₀
Burrows-Wheeler Transform: LF Mapping

BWM with T-ranking:

\[
F \quad L
\]

<table>
<thead>
<tr>
<th></th>
<th>a₀</th>
<th>b₀</th>
<th>a₁</th>
<th>a₂</th>
<th>b₁</th>
<th>a₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₃</td>
<td>$</td>
<td>a₀</td>
<td>b₀</td>
<td>a₁</td>
<td>a₂</td>
<td>b₁</td>
</tr>
<tr>
<td>a₁</td>
<td>a₂</td>
<td>b₁</td>
<td>a₃</td>
<td>$</td>
<td>a₀</td>
<td>b₀</td>
</tr>
<tr>
<td>a₂</td>
<td>b₁</td>
<td>a₃</td>
<td>$</td>
<td>a₀</td>
<td>b₀</td>
<td>a₁</td>
</tr>
<tr>
<td>a₀</td>
<td>b₀</td>
<td>a₁</td>
<td>a₂</td>
<td>b₁</td>
<td>a₃</td>
<td>$</td>
</tr>
<tr>
<td>b₁</td>
<td>a₃</td>
<td>$</td>
<td>a₀</td>
<td>b₀</td>
<td>a₁</td>
<td>a₂</td>
</tr>
<tr>
<td>b₀</td>
<td>a₁</td>
<td>a₂</td>
<td>b₁</td>
<td>a₃</td>
<td>$</td>
<td>a₀</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)
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 \texttt{a}s in this order relative to each other?

\[
\begin{array}{c}
\text{\$ a b a a b a}_3 \\
a_3 \text{ a b a a b a}_1 \\
a_1 \text{ a b a \$ a b}_0 \\
a_2 \text{ b a \$ a b a}_1 \\
a_0 \text{ b a a b a \$} \\
a_1 \text{ a \$ a b a a}_2 \\
a_0 \text{ a a b a \$ a}_0 \\
\end{array}
\]

They're sorted by right-context

\[
\begin{array}{c}
\text{\$ a b a a b a}_3 \\
a_3 \text{ a b a a b a}_1 \\
a_1 \text{ a b a \$ a b}_0 \\
a_2 \text{ b a \$ a b a}_1 \\
a_0 \text{ b a a b a \$} \\
a_1 \text{ a \$ a b a a}_2 \\
a_0 \text{ a a b a \$ a}_0 \\
\end{array}
\]

They're sorted by right-context

Why are these \texttt{a}s in this order relative to each other?

Occurrences of \texttt{c} in \texttt{F} are sorted by right-context. Same for \texttt{L}!

\textbf{Any ranking} we give to characters in \texttt{T} will match in \texttt{F} and \texttt{L}
Burrows-Wheeler Transform: LF Mapping

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

Given BWT = $a_3 \ b_1 \ b_0 \ a_1 \ \$ \ a_2 \ a_0$

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 \textit{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:

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>
<th>What is good about f-rank?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>$L$</td>
<td>$F$</td>
</tr>
<tr>
<td>$$$</td>
<td>$a_3$</td>
<td>$$$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$b_1$</td>
<td>$a_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_0$</td>
<td>$a_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$a_1$</td>
<td>$a_2$</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$$$</td>
<td>$a_3$</td>
</tr>
<tr>
<td>$b_1$</td>
<td>$a_2$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>$b_0$</td>
<td>$a_0$</td>
<td>$b_0$</td>
</tr>
</tbody>
</table>
Burrows-Wheeler Transform: A better ranking

$$T = \texttt{a b b c c d$}$$

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

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

Say $T$ has 300 As, 400 Cs, 250 Gs and 700 Ts 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 find any character in O(1) time!
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$
$a b a a b a$
$a b a a b a$
$a b a a b a$
$b a a b a a$
$b a a b a a$

BWM(T)

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

SA(T)
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

\[ P = \text{aba} \]

Easy to find all the rows beginning with \text{a}
FM Index: Querying

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

\[ P = aba \]

\[ F \quad L \]
\[
\begin{array}{c}
\$ \quad a \quad b \quad a \quad a \quad b \quad a_0 \\
a_0 \quad $ \quad a \quad b \quad a \quad a \quad b_0 \\
a_1 \quad a \quad b \quad a \quad $ \quad a \quad b_1 \\
a_2 \quad b \quad a \quad $ \quad a \quad b \quad a_1 \\
a_3 \quad b \quad a \quad a \quad b \quad a \quad $ \\
b_0 \quad a \quad $ \quad a \quad b \quad a \quad a_2 \\
b_1 \quad a \quad a \quad b \quad a \quad $ \quad a_3 \\
\end{array}
\]

\[ P = aba \]

\[ F \quad L \]
\[
\begin{array}{c}
\$ \quad a \quad b \quad a \quad a \quad b \quad a_0 \\
a_0 \quad $ \quad a \quad b \quad a \quad a \quad b \quad a_0 \\
a_1 \quad a \quad b \quad a \quad $ \quad a \quad b_1 \\
a_2 \quad b \quad a \quad $ \quad a \quad b \quad a_1 \\
a_3 \quad b \quad a \quad a \quad b \quad a \quad $ \\
b_0 \quad a \quad $ \quad a \quad b \quad a \quad a_2 \\
b_1 \quad a \quad a \quad b \quad a \quad $ \quad a_3 \\
\end{array}
\]

Look at those rows in $L$. $b_0, b_1$ are $b$s occurring just to left.

Use LF Mapping. Let new range delimit those $b$s

\textbf{Note:} We still aren’t storing the characters in grey, we just know they exist.
We have rows beginning with $\text{ba}$, now we seek rows beginning with $\text{aba}$.

\[ P = \text{aba} \]

We have rows beginning with $\text{ba}$, now we seek rows beginning with $\text{aba}$.

Now we have the rows with prefix $\text{aba}$. 
FM Index: Querying

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

$$P = \textbf{bba}$$

<table>
<thead>
<tr>
<th>$F$</th>
<th>$L$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$</td>
<td>a b a a b a_0</td>
</tr>
<tr>
<td>a_0</td>
<td>$$ a b a a b_0</td>
</tr>
<tr>
<td>a_1</td>
<td>a b a $ a b_1</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>

Rows with $\textbf{ba}$ prefix

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

<table>
<thead>
<tr>
<th>$F$</th>
<th>$L$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$</td>
<td>a b a a b $a_0$</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$$ a b a a b$ b_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>a b a $$ a b_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>b a $$ a b a_1$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>b a a b a $$</td>
</tr>
<tr>
<td>$b_0$</td>
<td>a $$ a b a a_2$</td>
</tr>
<tr>
<td>$b_1$</td>
<td>a a b a $$ a_3$</td>
</tr>
</tbody>
</table>

What are the values?
Burrows-Wheeler Transform

Reversible permutation of the characters of a string

\[
\begin{align*}
T & \quad \text{BWT}(T) \\
\end{align*}
\]

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

w$wwdd__nnoooaattTmmmrrrrrooo__ooo

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

s$esttssfftteww_hhmmboottttt_ii__woeeaaressIi_______

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

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.

<table>
<thead>
<tr>
<th>final char (L)</th>
<th>sorted rotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>n to decompress. It achieves compression</td>
</tr>
<tr>
<td>o</td>
<td>n to perform only comparisons to a depth</td>
</tr>
<tr>
<td>o</td>
<td>n transformation} This section describes</td>
</tr>
<tr>
<td>n</td>
<td>n transformation} We use the example and</td>
</tr>
<tr>
<td>t</td>
<td>n treats the right-hand side as the most</td>
</tr>
<tr>
<td>r</td>
<td>n tree for each 16 kbyte input block, enc</td>
</tr>
<tr>
<td>e</td>
<td>n tree in the output stream, then encodes</td>
</tr>
<tr>
<td>n</td>
<td>n turn, set $L[i]$ to be the</td>
</tr>
<tr>
<td>t</td>
<td>n turn, set $R[i]$ to the</td>
</tr>
<tr>
<td>o</td>
<td>n unusual data. Like the algorithm of Man</td>
</tr>
<tr>
<td>a</td>
<td>n use a single set of probabilities table</td>
</tr>
<tr>
<td>e</td>
<td>n using the positions of the suffixes in</td>
</tr>
<tr>
<td>i</td>
<td>n value at a given point in the vector $R</td>
</tr>
<tr>
<td>e</td>
<td>n when the block size is quite large. Ho</td>
</tr>
<tr>
<td>i</td>
<td>n which codes that have not been seen in</td>
</tr>
<tr>
<td>i</td>
<td>n with $ch$ appear in the {\em same order</td>
</tr>
<tr>
<td>i</td>
<td>n with $ch$. \hspace{1cm} In our exam</td>
</tr>
<tr>
<td>o</td>
<td>n with Huffman or arithmetic coding. Bri</td>
</tr>
<tr>
<td>o</td>
<td>n with figures given by Bell\cite{bell}.</td>
</tr>
</tbody>
</table>

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.

Burrows-Wheeler Transform

Lets compare the SA with the BWT...

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

$ a b a a b a
\[ a a b a a b \]
\[ a b a a b a \]
\[ a b a a b a \]
\[ a b a a b a \]
\[ b a a b a a \]
\[ b a a b a a \]
\[ b a a b a a \]
\[ b a a b a a \]

SA(T)

BWM(T)

Suffix Array is O(m)
Burrows-Wheeler Transform

Lets compare the SA with the BWT…

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

\begin{array}{c|c|c}
6 & & a \\
5 & & b \\
2 & & a \\
3 & & b \\
0 & & a \\
4 & & b \\
1 & & a \\
\end{array}

\begin{array}{c|c|c}
SA(T) & & BWT(T) \\
\text{Suffix Array is O(m)} & & \text{BWT is O(m)} \\
\end{array}

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

BWM is related to the suffix array

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

\[
\begin{align*}
6 & $ \\
5 & a $ \\
2 & a a b a $ \\
3 & a b a $ \\
0 & a b a a b a $ \\
4 & b a $ \\
1 & b a a b a $ \\
\end{align*}
\]

BWM(T) \quad SA(T)

Same order whether rows are rotations or suffixes
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”
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”