String Algorithms and Data Structures
Burrows-Wheeler Transform

CS 199-225
Brad Solomon

March 27, 2023

University of Illinois
Urbana-Champaign
Department of Computer Science
Informal Early Feedback

The instructor is well-prepared for each class / recording
10 responses

80% Strongly agree
20% Agree
Informal Early Feedback

I feel that I can actively participate in lecture
10 responses

- Strongly agree: 60%
- Agree: 30%
- Neutral: 10%

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

- Strongly agree: 50%
- Agree: 30%
- Neutral: 20%
Informal Early Feedback

I receive helpful and complete answers to my questions

During lecture

Outside lecture
Informal Early Feedback

Lecture helpfulness
- 60% Very helpful
- 30% Helpful
- 10% Not helpful

Assignment helpfulness
- 50% Very helpful
- 40% Helpful
- 10% Not helpful
Informal Early Feedback

I think it was very useful in learning no necessarily string algorithms themselves, but algorithms in general. It feels like a good introduction to how to optimize a seemingly simple naive problem.

Learning different interesting algorithms for operation on string help me to have better ability to handle problems.

I am not sure what group activity could be added but something that allows us to interact with other students more.

I wouldn't mind having some segments that goes deeper into the material and theory covered in the parent CS 225 course e.g. fuller implementations of DSAs and their associated algorithms. But string matching is pretty cool as well!
Exact pattern matching \textit{w/ indexing}

There are many data structures built on 	extit{suffixes}

We have now seen both of these data structures
Exact pattern matching \textit{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>

$m = |T|$, $n = |P|$, $k = \# \text{ occurrences of } P \text{ in } T$
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)!
Exact pattern matching \textit{w/ indexing}

The basis of the FM index is a \textit{transformation}
Burrows-Wheeler Transform

*Reversible permutation* of the characters of a string

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 $
  T
```

### All rotations

```
  ????
```
Text rotations

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

(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

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

\[
\begin{align*}
\text{a b a a b a $} & \quad \text{BWT(T)} \\
\text{Sort} & \quad \text{Last column}
\end{align*}
\]

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

Last column

$T = \text{c a r }$
$
\text{a r }\text{c}$
$
\text{c a r }$
$
\text{r }\text{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?

$abaabab$a

Sort

$abaabab$a

Burrows-Wheeler Matrix

Last column

abaababa$a

BWT(T)

All rotations

abaababa$a

abaababa$a

abaababa$a

abaababa$a

abaababa$a

abaababa$a
Burrows-Wheeler Transform

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

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

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

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

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

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

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

What is the right context of apple? le $ap$

A letter always has the same right context.

```
$ apple
apple$
apple $
$e apple
$e apple
$p l e $ap
$ple ap
$pp ple$ a
```
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.

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

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

BWM with T-ranking:

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

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

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

Same with $b$s: $b₁, b₀$
Burrows-Wheeler Transform: LF Mapping

BWM with T-ranking:

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

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 \textbf{as} in this order relative to each other?

\[ \begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c} \hline & \$: & a & b & a & a & b & a & 3 \\
\hline a_3 & $ & a & b & a & a & b & a & 1 \\
\hline a_1 & a & b & a & $ & a & b & a & 0 \\
\hline a_2 & b & a & $ & a & b & a & 1 \\
\hline a_0 & b & a & b & a & $ \\
\hline b_1 & a & $ & a & b & a & a & 2 \\
\hline b_0 & a & a & b & a & $ & a & 0 \\
\hline \end{array} \]

They're sorted by right-context

\[ \begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c} \hline & \$: & a & b & a & a & b & a & 3 \\
\hline a_3 & $ & a & b & a & a & b & a & 1 \\
\hline a_1 & a & b & a & $ & a & b & a & 0 \\
\hline a_2 & b & a & $ & a & b & a & 1 \\
\hline a_0 & b & a & b & a & $ \\
\hline b_1 & a & $ & a & b & a & a & 2 \\
\hline b_0 & a & a & b & a & $ & a & 0 \\
\hline \end{array} \]

They're sorted by right-context

Why are these \textbf{as} in this order relative to each other?

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

\textbf{Any ranking} we give to characters in \textit{T} will match in \textit{F} and \textit{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 beginning with $a_3$. $L$ contains character just prior to $a_3$: $b_1$.

Repeat for $b_1$, get $a_2$
Repeat for $a_2$, get $a_1$
Repeat for $a_1$, get $b_0$
Repeat for $b_0$, get $a_0$
Repeat for $a_0$, 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]
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$
$a \ a \ b \ a \ a \ b$
$a \ a \ b \ a \ a \ b$
$b \ a \ a \ b \ a \ a$
$a \ b \ a \ a \ b \ a \ a$
$a \ a \ b \ a \ a \ b \ a \ a$

BWM(T)

<table>
<thead>
<tr>
<th></th>
<th>$</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>$</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>a</td>
<td>$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

$P = \text{aba}$

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

We have rows beginning with \( a \), now we want rows beginning with \( ba \)

\[ P = aba \]

\[ F \]

\[ L \]

\begin{align*}
\$ & a b a a b a_0 \\
a_0 & b a a a b_0 \\
a_1 & a b a a a_b_1 \\
a_2 & b a $ a b a_1 \\
a_3 & b a b a a $ \\
b_0 & a $ a b a a_2 \\
b_1 & a a b a a $ a_3 \\
\end{align*}

\[ P = aba \]

\[ F \]

\[ L \]

\begin{align*}
\$ & a b a a b a_0 \\
a_0 & $ a b a a b_0 \\
a_1 & a b a a a b_1 \\
a_2 & b a $ a b a_1 \\
a_3 & b a b a a $ \\
b_0 & a $ a b a a_2 \\
b_1 & a a b a a $ a_3 \\
\end{align*}

Look at those rows in \( L \).

\( b_0, b_1 \) are \( bs \) occurring just to left.

Use LF Mapping. Let new range delimit those \( bs \).

\[ b_0 \]

\[ b_1 \]

\textbf{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}

\[ P = \text{aba} \]

\[ \begin{array}{ll}
F & L \\
\$
& a \\
a_0 
& a
\\a_1 
& b \\
a_2 
& b \\
a_3 
& b \\
\hline
\text{b}_0 
& a \\
\text{b}_1 
& a
\end{array} \]

\[ F \]

\[ \begin{array}{ll}
F & L \\
\$
& a \\
a_0 
& a \\
a_1 
& b \\
a_2 
& b \\
a_3 
& b \\
\hline
\text{b}_0 
& a \\
\text{b}_1 
& a
\end{array} \]

\[ L \]

\textbf{Use LF Mapping}

\[ P = \text{aba} \]

\[ \begin{array}{ll}
F & L \\
\$
& a \\
a_0 
& a \\
a_1 
& b \\
a_2 
& b \\
a_3 
& b \\
\hline
\text{b}_0 
& a \\
\text{b}_1 
& a
\end{array} \]

\textbf{P = aba}

\[ \begin{array}{ll}
F & L \\
\$
& a \\
a_0 
& a \\
a_1 
& b \\
a_2 
& b \\
a_3 
& b \\
\hline
\text{b}_0 
& a \\
\text{b}_1 
& a
\end{array} \]

\textbf{Now we have the rows with prefix \textbf{aba}}

\[ a_2, a_3 \text{ occur just to left.} \]
FM Index: Querying

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

\[
P = \text{bba}
\]

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

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

No bs!
If we scan characters in the last column, that can be slow, \(O(m)\)

**Problem 1:** If we _scan_ characters in the last column, that can be slow, \(O(m)\)

\[ P = \text{aba} \]

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>$</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
</tbody>
</table>

Scan, looking for bs
PFM 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

<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 $$$ a b $b_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>b a $$$ a b a 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>

Where are the values?

[3, 5)
Burrows-Wheeler Transform

Reversible permutation of the characters of a string

$$T \quad \text{BWT}(T)$$

B A N A N A $ \leftrightarrow A N N B $ A A

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_and_tomorrow$

tomow$tomorrow_and_tomorrow_and_tomorrow_and_tomorrow$

tomorrow_and_tomorrow$tomorrow_and_tomorrow_and_tomorrow$

tomorrow_and_tomorrow_and_tomorrow_and_tomorrow$

tomorrow_and_tomorrow_and_tomorrow_and_tomorrow$

tomorrow_and_tomorrow_and_tomorrow_and_tomorrow$

tomorrow_and_tomorrow_and_tomorrow_and_tomorrow$

tomorrow_and_tomorrow_and_tomorrow_and_tomorrow$

tomorrow_and_tomorrow_and_tomorrow_and_tomorrow$

tomorrow_and_tomorrow_and_tomorrow$

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</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>n</td>
<td>n treats the right-hand side as the most</td>
</tr>
<tr>
<td>n</td>
<td>n tree for each 16 kbyte input block, enc</td>
</tr>
<tr>
<td>a</td>
<td>n tree in the output stream, then encodes</td>
</tr>
<tr>
<td>a</td>
<td>n turn, set $L[i]$ to be the</td>
</tr>
<tr>
<td>i</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>e</td>
<td>n value at a given point in the vector $R$</td>
</tr>
<tr>
<td>e</td>
<td>n we present modifications that improve t</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 $\text{ch}$ appear in the \text{same order}</td>
</tr>
<tr>
<td>i</td>
<td>n with $\text{ch}$. 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

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

$T = a b a a b a$

$SA(T)$

$BWM(T)$

Suffix Array is $O(m)$
Burrows-Wheeler Transform

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

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

<table>
<thead>
<tr>
<th></th>
<th>SA(T)</th>
<th>BWT(T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td>$\text{ab}$</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>$\text{a}$</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>$\text{ab}$</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>$\text{b}$</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>$\text{a}$</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>$\text{a}$</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>$\text{a}$</td>
</tr>
</tbody>
</table>

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
Burrows-Wheeler Transform

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

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

“\( \text{BWT} = \text{characters just to the left of the suffixes in the suffix array} \)”

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

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

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

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

“$BWT = \text{characters just to the left of the suffixes in the suffix array}$”