The Burrows-Wheeler Transform and Bioinformatics

J. Matthew Holt
holtjma@cs.unc.edu
Last Class - Multiple Pattern Matching Problem

$m$ - length of text  
$d$ - max length of pattern  
$x$ - number of patterns

<table>
<thead>
<tr>
<th>Method</th>
<th>Storage Cost</th>
<th>Single Pattern Search Time</th>
<th>Multiple Pattern Search Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute Force</td>
<td>$O(m)$</td>
<td>$O(dm)$</td>
<td>$O(xdm)$</td>
</tr>
<tr>
<td>Keyword Tries</td>
<td>$O(xd)$</td>
<td>$O(dm)$</td>
<td>$O(dm)$</td>
</tr>
<tr>
<td>Suffix Trees</td>
<td>$O(m)$ [20m bytes]</td>
<td>$O(d)$</td>
<td>$O(xd)$</td>
</tr>
<tr>
<td>Suffix Arrays</td>
<td>$O(m^*\log(m))$ [4m bytes]</td>
<td>$O(d^*\log(m))$</td>
<td>$O(xd^*\log(m))$</td>
</tr>
<tr>
<td>BWT</td>
<td>$O(m)$ [often $m$ bits]</td>
<td>$O(d)$</td>
<td>$O(xd)$</td>
</tr>
</tbody>
</table>
Recall Suffix Arrays

- Create all suffix - note we are doing **cyclic** suffixes now
- Sort the suffixes
- Indices are stored

<table>
<thead>
<tr>
<th>Index (N)</th>
<th>Rotations</th>
<th>Suffix Array</th>
<th>Sorted Suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ACACGGACA$</td>
<td>9</td>
<td>$ACACGGACA</td>
</tr>
<tr>
<td>1</td>
<td>CACGGACA$A</td>
<td>8</td>
<td>A$ACACGGAC</td>
</tr>
<tr>
<td>2</td>
<td>ACGGACA$AC</td>
<td>6</td>
<td>ACA$ACACGG</td>
</tr>
<tr>
<td>3</td>
<td>CGGAC$ACA</td>
<td>0</td>
<td>ACACGGACA$</td>
</tr>
<tr>
<td>4</td>
<td>GGAC$ACAC</td>
<td>2</td>
<td>ACGGACA$AC</td>
</tr>
<tr>
<td>5</td>
<td>GAC$ACACG</td>
<td>7</td>
<td>CA$ACACGG</td>
</tr>
<tr>
<td>6</td>
<td>A$ACACGG</td>
<td>1</td>
<td>CACGGACA$A</td>
</tr>
<tr>
<td>7</td>
<td>CA$ACACGGA</td>
<td>3</td>
<td>CGGAC$ACA</td>
</tr>
<tr>
<td>8</td>
<td>A$ACACGGAC</td>
<td>5</td>
<td>GAC$ACACG</td>
</tr>
<tr>
<td>9</td>
<td>$ACACGGACA</td>
<td>4</td>
<td>GGAC$ACAC</td>
</tr>
</tbody>
</table>

The suffix array for string “ACACGGACA$”
“$” is just an end-of-string character
Recall Suffix Arrays (cont.)

- $N$ - number of bases (length of text)
- $k$ - pattern length (also called a $k$-mer)
- Space complexity: $O(N \cdot \log(N))$ bits
  - Stored as offsets into original string
  - $N$ offsets that require $\log(N)$ bits per value
- Search time: $O(k \cdot \log(N))$ operations
  - Binary search require $O(\log(N))$ string comparisons
  - Each string comparison requires $O(k)$ symbol comparisons
- Problem:
  - $O(k \cdot \log(N))$ can be large when strings are billions of characters long
The Burrows-Wheeler Transform

- Burrows & Wheeler, 1994
- **BWT** - a permutation of a string that implicitly represent a suffix array of the same string
- Naive construction:
  - Create all suffixes
  - Sort suffixes
  - Concatenate last symbols in suffixes
- Implicit suffix array
  - “Last symbol” in suffix
  - “Previous symbol” to suffix

<table>
<thead>
<tr>
<th>Index (N)</th>
<th>Rotations</th>
<th>Sorted Suffixes</th>
<th>BWT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ACACGGACA$</td>
<td>$ACACGGACA</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>CACGGACA$A</td>
<td>A$ACACGGAC</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>ACGGACA$AC</td>
<td>ACA$ACACGG</td>
<td>G</td>
</tr>
<tr>
<td>3</td>
<td>CGGACA$ACA</td>
<td>ACACGGACA$</td>
<td>$</td>
</tr>
<tr>
<td>4</td>
<td>GGACA$ACAC</td>
<td>ACGGACA$AC</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>GACA$ACACG</td>
<td>CA$ACACGGA</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>ACA$ACACGG</td>
<td>CACGGACA$A</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>CA$ACACGGA</td>
<td>CGGACA$AC</td>
<td>A</td>
</tr>
<tr>
<td>8</td>
<td>A$ACACGGAC</td>
<td>GACA$ACACG</td>
<td>G</td>
</tr>
<tr>
<td>9</td>
<td>$ACACGGACA</td>
<td>GGACA$ACAC</td>
<td>C</td>
</tr>
</tbody>
</table>
BWT algorithm

BWT (string text)

\[ \text{table}_i = \text{Rotate(text, i)} \text{ for } i = 0..\text{len(text)}-1 \]

sort table alphabetically

return (last column of the table)

\[
\begin{align*}
tarheel$ & \quad $tarheel \\
arheel$t & \quad arheel$t \\
rheel$ta & \quad eel$starh \\
heel$tar & \quad el$tarhe \\
eel$tarh & \quad heel$tar \\
el$tarhe & \quad l$tarhee \\
l$tarhee & \quad rheel$ta \\
$tarheel & \quad tarheel$
\end{align*}
\]

BWT(“tarheels$”) = “ltherea$”
Example

BWT (string text)

\[
\text{table}_i = \text{Rotate}(\text{text}, i) \text{ for } i = 0..\text{len}(\text{text})-1
\]

sort table alphabetically

return (last column of the table)

\[
\begin{align*}
\text{banana}\$ \\
& \downarrow \downarrow \\
& a\$\text{banan} \\
& \downarrow \downarrow \\
& \text{ana}\$\text{ban} \\
& \downarrow \downarrow \\
& \text{anana}\$\text{b} \\
& \downarrow \downarrow \\
& \text{banana}\$ \\
& \downarrow \downarrow \\
& \text{na}\$\text{bana} \\
& \downarrow \downarrow \\
& \text{nana}\$\text{ba}
\end{align*}
\]

BWT = “annb$aa”
BWT in Python

**t** - a text that we want a BWT of

```python
def BWT(t):
    #create a list of all cyclic suffixes of t
    rotation = [t[i:]+t[:i] for i in xrange(len(t))]
    #sort the suffixes
    rotation.sort()
    #concatenate the last symbol from each suffix
    return "".join(r[-1] for r in rotation)
```

**sa** - the suffix array for text **t** (see last slides)

```python
def BWT_fromSA(t, sa):
    return "".join(t[v-1] for v in sa)
```
Inverting a BWT

- A property of a transform is that there is no information loss and they are invertible.

```plaintext
inverseBWT(string s)
    add $s$ as the first column of a table strings
    repeat length(s)-1 times:
        sort rows of the table alphabetically
        add $s$ as the first column of the table
    return (row that ends with the 'EOF' character)
```
def inverseBWT(t):
    #initialize the table from t
    table = [c for c in t]
    for j in xrange(len(t)-1):
        #sort the table
        table.sort()
        #insert the BWT as the first column
        table = [t[i]+table[i] for i in xrange(len(t))]
    #return the row that ends with end-of-string ‘$’
    return table[[r[-1] for r in table]].index('$$')]}
BWT Compression

- Uncompressed
  - Same as input text size
  - $O(N)$ bytes
- Compression
  - Tendency to form long runs
  - Run-length encoding (RLE)
- Can be stored as: ACG$C3AGC

<table>
<thead>
<tr>
<th>Index (N)</th>
<th>Suffix Array</th>
<th>BWT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$ACACGGAC\text{A}$</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>A$ACACGG\text{A}\text{C}$</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>ACA$ACACG\text{G}$</td>
<td>G</td>
</tr>
<tr>
<td>3</td>
<td>ACACGGACA$\text{S}$</td>
<td>$</td>
</tr>
<tr>
<td>4</td>
<td>ACGGACA$\text{S}\text{A}$</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>CA$ACACG\text{G}G\text{A}$</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>CACGGACA$\text{S}\text{A}$</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>CGGACA$\text{S}\text{A}\text{C}$</td>
<td>A</td>
</tr>
<tr>
<td>8</td>
<td>GACA$ACAC\text{G}$</td>
<td>G</td>
</tr>
<tr>
<td>9</td>
<td>GGACA$\text{S}\text{A}\text{C}$</td>
<td>C</td>
</tr>
</tbody>
</table>
Why does it form runs?

- Think about a BWT and suffix array of a book such as “Green Eggs and Ham”

<table>
<thead>
<tr>
<th>Sorted Suffixes</th>
<th>BWT</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Would you eat them in a box? ...</td>
<td>W</td>
</tr>
<tr>
<td>Would you eat them with a fox? ...</td>
<td>W</td>
</tr>
<tr>
<td>Would you like them here or there? ...</td>
<td>W</td>
</tr>
<tr>
<td>Would you like them in a house? ...</td>
<td>W</td>
</tr>
<tr>
<td>Would you like them with a mouse? ...</td>
<td>W</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Assumption is that patterns that cluster together will have the same symbol preceding them; thus the BWT forms long runs
FM-Index

- Ferragina & Manzini, 2005
- Enables fast exact searches
- **LF-mapping property** - Takes advantage of “last-first” relationship between BWT and suffix array
  - See colors on right
  - First “A” in BWT corresponds to first suffix starting with “A”
  - First “C” in BWT corresponds to first suffix starting with “C”
  - …
  - Second “A” in BWT corresponds to second suffix starting with “A”
  - …

<table>
<thead>
<tr>
<th>Index (N)</th>
<th>Suffix Array</th>
<th>BWT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$ACACGGGACA</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>A$ACACGGGAC</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>ACAGGGAC</td>
<td>G</td>
</tr>
<tr>
<td>3</td>
<td>ACACGGGAC$</td>
<td>$</td>
</tr>
<tr>
<td>4</td>
<td>ACAGGGAC$AC</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>CA$ACACGGGA</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>CACGGGAC$A</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>CGGACASACAC</td>
<td>A</td>
</tr>
<tr>
<td>8</td>
<td>GACASACACG</td>
<td>G</td>
</tr>
<tr>
<td>9</td>
<td>GGACASACAC</td>
<td>C</td>
</tr>
</tbody>
</table>
**FM-index (cont.)**

- **A** - alphabet size
- **N** - text length
- **F** - FM-index
  - $F[i][c]$ stores the number of times symbol $c$ occurs before index $i$
  - $O(NA)$ memory
  - Generated in a linear pass over the BWT
- **O** - Offset Array
  - $O[c]$ stores the index of the first suffix starting with symbol $c$
  - Derived from the final entry in $F$
  - $O(A)$ memory

<table>
<thead>
<tr>
<th>Index (N)</th>
<th>Suffix Array (not stored)</th>
<th>BWT</th>
<th>FM-index (F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\text{ACACGGAC}A$</td>
<td>A</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>$\text{A}$ACACGGAC$C$</td>
<td>C</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>2</td>
<td>$\text{ACA}$ACACGG$G$</td>
<td>G</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>3</td>
<td>ACACGGACA$$</td>
<td>$S$</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>4</td>
<td>ACGGACA$$A$C$</td>
<td>C</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>5</td>
<td>CA$$ACACGG$A$</td>
<td>A</td>
<td>1 1 2 1</td>
</tr>
<tr>
<td>6</td>
<td>CACGGACA$$A$</td>
<td>A</td>
<td>1 2 2 1</td>
</tr>
<tr>
<td>7</td>
<td>CCGACA$$A$CA$</td>
<td>A</td>
<td>1 3 2 1</td>
</tr>
<tr>
<td>8</td>
<td>GAC$$ACAC$G$</td>
<td>G</td>
<td>1 4 2 1</td>
</tr>
<tr>
<td>9</td>
<td>GGACA$$ACAC$C$</td>
<td>C</td>
<td>1 4 2 2</td>
</tr>
<tr>
<td>10</td>
<td>—</td>
<td>—</td>
<td>1 4 3 2</td>
</tr>
</tbody>
</table>

Offset (O): — 0 1 5 8
Find Predecessor Suffix

- Given an index \( i \) in the BWT, find the index in the BWT of the suffix preceding the suffix represented by \( i \)
  - suffix 0 is preceded by suffix 1
  - suffix 1 is preceded by suffix 5
  - suffix 5 is preceded by suffix 2
- The predecessor suffix of index \( i \):
  \( c = \text{BWT}[i] \)
  \( \text{predec} = O[c] + F[i][c] \)
- Predecessor of index 1
  \( c = \text{BWT}[1] = 'C' \)
  \( \text{predec} = O['C'] + F[1]['C'] = 5 + 0 = 5 \)
- Predecessor of index 8
  \( c = \text{BWT}[8] = 'G' \)
  \( \text{predec} = O['G'] + F[8]['G'] = 8 + 1 = 9 \)
- Time to find predecessor: \( O(1) \)

<table>
<thead>
<tr>
<th>Index (N)</th>
<th>Suffix Array (not stored)</th>
<th>BWT</th>
<th>FM-index (F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$ACACGGAC$A</td>
<td>A</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>A$ACACGGAC$c</td>
<td>C</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>2</td>
<td>ACA$ACACGGG$G</td>
<td>G</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>3</td>
<td>ACACGGAC$A</td>
<td>$</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>4</td>
<td>ACGGACA$A$c</td>
<td>C</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>5</td>
<td>CA$ACACGGG$A</td>
<td>A</td>
<td>1 1 2 1</td>
</tr>
<tr>
<td>6</td>
<td>CACGGAC$A</td>
<td>A</td>
<td>1 2 2 1</td>
</tr>
<tr>
<td>7</td>
<td>CGGAC$A$AC</td>
<td>A</td>
<td>1 3 2 1</td>
</tr>
<tr>
<td>8</td>
<td>GACA$ACACG$G</td>
<td>G</td>
<td>1 4 2 1</td>
</tr>
<tr>
<td>9</td>
<td>GGAC$A$ACAC</td>
<td>C</td>
<td>1 4 2 2</td>
</tr>
<tr>
<td>10</td>
<td>—</td>
<td>—</td>
<td>1 4 3 2</td>
</tr>
<tr>
<td>Offset (O)</td>
<td>—</td>
<td>—</td>
<td>0 1 5 8</td>
</tr>
</tbody>
</table>
Suffix Recovery

- Suffix recovery:
  - Start at an index $i$
  - Repeatedly find the predecessor
  - Stop when back at original index
- Original string recovery:
  - Start at 0
  - Repeatedly find predecessor until back at 0
  - $O(N)$ time to get original string back

```python
def recoverSuffix(BWT, F, O, i):
    ret = []
    c = BWT[i]
    predec = O[c]+F[i][c]
    ret.append(c)
    while predec != i:
        c = BWT[predec]
        predec = O[c]+F[predec][c]
        ret.append(c)
    return "".join(ret[::-1])
```
Find $k$-mer

- **$k$-mer**: a pattern of length $k$
- All searches occur in reverse order
  - Start with full BWT range (0, N)
  - Restrict by one symbol at a time
- Find $k$-mer “ACA”
  - Initialize to full range ("")
    - **low**, **high** = 0, 10
  - Find occurrences of “A”
    - **low** = O[‘A’]+F[low][‘A’] = 1+0 = 1
    - **high** = O[‘A’]+F[high][‘A’] = 1+4 = 5
  - Find occurrences of “CA”
    - **low** = O[‘C’]+F[low][‘C’] = 5+0 = 5
    - **high** = O[‘C’]+F[high][‘C’] = 5+2 = 7
  - Find occurrences of “ACA”
    - **low** = O[‘A’]+F[low][‘A’] = 1+1 = 2
    - **high** = O[‘A’]+F[high][‘A’] = 1+3 = 4

<table>
<thead>
<tr>
<th>Index (N)</th>
<th>Suffix Array (not stored)</th>
<th>BWT</th>
<th>FM-index (F)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>0</td>
<td>$ACACGGAC$A</td>
<td>A</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>A$ACACGGAC$C</td>
<td>C</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>2</td>
<td>ACA$ACACGGG$G</td>
<td>G</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>3</td>
<td>ACACGGACA$S</td>
<td>$</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>4</td>
<td>ACGGACA$AC$C</td>
<td>C</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>5</td>
<td>CA$ACACGGG$A</td>
<td>A</td>
<td>1 1 2 1</td>
</tr>
<tr>
<td>6</td>
<td>CACGGACASA$A</td>
<td>A</td>
<td>1 2 2 1</td>
</tr>
<tr>
<td>7</td>
<td>CGGACA$ACA$A</td>
<td>A</td>
<td>1 3 2 1</td>
</tr>
<tr>
<td>8</td>
<td>GACA$ACACG$G</td>
<td>G</td>
<td>1 4 2 1</td>
</tr>
<tr>
<td>9</td>
<td>GGACA$ACAC$C</td>
<td>C</td>
<td>1 4 2 2</td>
</tr>
<tr>
<td>10</td>
<td>—</td>
<td></td>
<td>1 4 3 2</td>
</tr>
</tbody>
</table>

Offset (O) — 0 1 5 8
Find \( k \)-mer (cont.)

- \( p \) - pattern
- \( F \) - FM-index
- Time complexity - \( O(k) \)
  - Requires \( O(k) \) lookups
  - Search time only dependent on length of \( k \)-mer
  - **Does not** depend on BWT (data) size!!!

```python
def find(p, F, O):
    lo = 0
    hi = len(F)
    for l in reversed(p):
        lo = O[l] + F[lo][l]
        hi = O[l] + F[hi][l]
    return lo, hi
```
def find(p, F, O):
    lo = 0
    hi = len(F)
    for l in reversed(p):
        lo = O[l] + F[lo][l]
        hi = O[l] + F[hi][l]
    return lo, hi

find("AGG", F, 0)
Practical Adaptations

- Compressed BWT is small
- FM-index is not, \( O(A^N) \) for alphabet of size \( A \) and a BWT of length \( N \)
- Trade-off, space v. time:
  - Use a sampled FM-index
  - \( B \) - bin size
  - Uses \( O(A^N/B) \) values
  - Requires \( O(B) \) time per lookup (for a fixed size \( B \), this is just a larger constant time lookup)
  - I typically use 1024 in practice
Tools using BWTs in Exact Pattern Matching

- Alignment
  - Bowtie (2009) and BWA (2009)
  - Build a BWT of the reference genome (~2-3 GB)
  - Align:
    - Given a 100 base pair read
    - Cut into smaller seed pieces (i.e. four 25-mers)
    - Exact search for the pieces separately - very fast using BWT
    - Use local alignment (dynamic program) to extend initial seed alignments and account for errors
  - Bowtie2 (2011) and Tophat2 (2013) are still very prominent and fast aligners
BWTs and String Collections

- We have a BWT of single string with some functions
  - Recover suffix
  - $O(k)$ search for an arbitrary $k$-mer
- What if you have multiple strings?
  - Concatenate strings together?
  - Build each one as a separate BWT and query each one?
Multi-string BWTs

- **MSBWT** - a BWT containing a string collection instead of just a single string
- Earliest: Mantaci *et al.* (2005), used concatenation approach
- Bauer *et al.* (2011) - proposed version we will discuss today
MSBWT Construction

- Naive Construction:
  - Create all rotations for all strings in the collection
  - Sort all rotations together (Suffix Array)
  - Store the last symbols in each suffix
- Strings are "cyclic"
  - Getting the predecessor always gets a suffix from the same string
  - Impossible to "jump" from one string to another

<table>
<thead>
<tr>
<th>Index</th>
<th>Rotations</th>
<th>Sorted Suffixes</th>
<th>MSBWT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ACCA$</td>
<td>$ACCA</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>CCA$A</td>
<td>$CAA$</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>CA$AC</td>
<td>A$ACC</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>A$ACC</td>
<td>A$CAA</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>$ACCA</td>
<td>AA$CA</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>CAAAA$</td>
<td>AAA$C</td>
<td>C</td>
</tr>
<tr>
<td>6</td>
<td>AAA$C</td>
<td>ACCA$</td>
<td>$</td>
</tr>
<tr>
<td>7</td>
<td>AA$CA</td>
<td>CA$AC</td>
<td>C</td>
</tr>
<tr>
<td>8</td>
<td>A$CAA</td>
<td>CAAA$</td>
<td>$</td>
</tr>
<tr>
<td>9</td>
<td>$CAAA</td>
<td>CCA$A</td>
<td>A</td>
</tr>
</tbody>
</table>

The multi-string BWT for strings "ACCA$" and "CAAA$".
MSBWT and FM-index

- Identical Definition
- Find $k$-mer “CA”
  - Initialize to full range ("")
    low, high = 0, 10
  - Find occurrences of “A”
    low = $O[\text{‘A’}]+F[\text{low}][\text{‘A’}] = 2+0 = 2$
    high = $O[\text{‘A’}]+F[\text{high}][\text{‘A’}] = 2+5 = 7$
  - Find occurrences of “CA”
    low = $O[\text{‘C’}]+F[\text{low}][\text{‘C’}] = 7+0 = 7$
    high = $O[\text{‘C’}]+F[\text{high}][\text{‘C’}] = 7+2 = 9$

<table>
<thead>
<tr>
<th>Index</th>
<th>Suffix Array</th>
<th>MSBWT</th>
<th>FM-index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>0</td>
<td>$ACC\text{A}$</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$C\text{AAA\text{A}}$</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>A$\text{ACC\text{C}}$</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>A$\text{CAA\text{A}}$</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>AA$\text{C\text{A\text{A}}}$</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>AAA$\text{C\text{C}}$</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>ACCA$\text{$\text{S}$}$</td>
<td>$\text{S}$</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>CA$\text{$\text{A\text{C}}$}$</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>CAAA$\text{$\text{S}$}$</td>
<td>$\text{S}$</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>CCA$\text{$\text{A\text{A}}$}$</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>—</td>
<td>—</td>
<td>2</td>
</tr>
<tr>
<td>Offset (O)</td>
<td>—</td>
<td>—</td>
<td>0</td>
</tr>
</tbody>
</table>
Compression of sequencing datasets

- Using Run-length encoding again
- Reasons we expect compression:
  - True genomic repeats: gene families, long repeats, etc.
  - Over-sampling: 30x coverage means we expect 30 copies of every \( k \)-mer pattern
  - Sequencing errors may break up runs
  - Technical errors may cause biases for or against a particular pattern
- Real Mouse DNA-seq:
  - ~350 Giga-bases
  - ~41 GB using RLE (0.94 bits/base)
- Real Mouse RNA-seq:
  - ~8.9 Giga-bases
  - ~1.2 GB using RLE (1.05 bits/base)
MSBWT Applications

- Instead of building a BWT of a reference genome, build a MSBWT of the sequenced reads
- Arbitrary exact match $k$-mer queries
  - $O(k)$ time
  - Enables fast searches/counting
- Recover an arbitrary read of length $L$ from MSBWT
  - $O(L)$ time
  - Enables extraction of user-selected reads
K-mer Search & Extraction

- Basic utilization
- Search for all reads with a given $k$-mer
- Extract all reads with that $k$-mer or the reverse-complement of the $k$-mer
- Build a consensus
Reference-based Searches

- Given a reference genome and region of that genome
- Split reference into $k$-mers
- Count the abundance of each $k$-mer and plot
  - Fast - $O(k)$ time per $k$-mer
  - Similar to a post-alignment pileup

CAST/EiJ at *Egr3*, counting 40-mers overlapping by 20
CAST/EiJ DNA-seq for annotated gene *Igf2*

Uncorrected

149,838,013: 0  TTGATGGCTCGATGCATTCA
149,838,033: 0  TTACCTGATCAGTCTCCCGTTATGAGGAATGGGTACA

Corrected

149,838,013: 18 TTGATGGCTCGATGCATTCTTCACCTGATCAGTCTCCCG
149,838,033: 17 TTACCTGATCAGTCTCCCGTTATGAGGAATGGGTACA
Targeted Assembly

- *De novo* assembly given a *k*-mer target known as the “seed” *k*-mer
- Extend the seed by counting the occurrence of each possible extension
- Generates a graph extending from the seed
  - Nodes - continuous unambiguous choice of extensions (similar to a contig)
  - Edges - multiple possible choices for extension
Targeted Assembly Tool Demo

- Demo used for CS events:
  - http://www.csbio.unc.edu/CEGSseq/index.py?run=msDemoTarget
- Gene
- Mitochondria
Summary

• Burrows-Wheeler Transform
  • Permutation of characters that represents a suffix array
  • Run-length encoded for compression
• FM-index
  • Derived from BWT
  • Exploits LF-mapping property
  • \(O(k)\) search time for arbitrary \(k\)-mer
  • Used in many fast aligners
• MSBWT
  • Applies to string collections
  • Enables database-like access to reads via \(k\)-mer searches