CSE 549: BWT & FM-Index

All slides in this lecture not marked with “*” courtesy of Ben Langmead.
Burrows-Wheeler Transform

Reversible permutation of the characters of a string, used originally for compression

How is it useful for compression?  How is it reversible?  How is it an index?

Burrows-Wheeler Transform

```python
def rotations(t):
    """ Return list of rotations of input string t """
    tt = t * 2
    return [ tt[i:i+len(t)] for i in xrange(0, len(t)) ]

def bwm(t):
    """ Return lexicographically sorted list of t’s rotations """
    return sorted(rotations(t))

def bwtViaBwm(t):
    """ Given T, returns BWT(T) by way of the BWM """
    return ''.join(map(lambda x: x[-1], bwm(t)))

>>> bwtViaBwm("Tomorrow_and_tomorrow_and_tomorrow$")
'w$wwdd__nnnoooaattTmmmrrrrrooo_ooo'

>>> bwtViaBwm("It_was_the_best_of_times_it_was_the_worst_of_times$")
's$esttssfftteww_hhmmboottttt_iii_woeeaaressIi______'

>>> bwtViaBwm('in_the_jingle_jangle_morning_Ill_come_following_you$')
'u_gleeeengj_mlhl_nnnnt$nwj__lggIolo_iiiiarfcmypo_oo_'
```

Python example: [http://nbviewer.ipython.org/6798379](http://nbviewer.ipython.org/6798379)
Burrows-Wheeler Transform

Characters of the BWT are sorted by their right-context
This lends additional structure to BWT(T), tending to make it 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.
Burrows-Wheeler Transform

BWM bears a resemblance to the suffix array

BWM(T)

\[
\begin{array}{l}
\$ \ a \ b \ a \ a \ b \ a \\
 a \ $ \ a \ b \ a \ a \ b \\
a \ a \ b \ a \ $ \ a \ b \\
a \ b \ a \ $ \ a \ b \ a \\
a \ b \ a \ a \ b \ a \ $ \\
b \ a \ $ \ a \ b \ a \ a \\
b \ a \ a \ b \ a \ $ \ a \\
\end{array}
\]

SA(T)

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

Sort order is the same 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”
def suffixArray(s):
    """ Given T return suffix array SA(T). We use Python's sorted function here for simplicity, but we can do better. """
    satups = sorted(((s[i:], i) for i in xrange(0, len(s))))
    # Extract and return just the offsets
    return map(lambda x: x[1], satups)

def bwtViaSa(t):
    """ Given T, returns BWT(T) by way of the suffix array. """
    bw = []
    for si in suffixArray(t):
        if si == 0: bw.append('$')
        else: bw.append(t[si-1])
    return ''.join(bw) # return string-ized version of list bw

>>> bwtViaSa("Tomorrow_and_tomorrow_and_tomorrow$")
'w$wwdd__nnnoaaattTmmmrrrrrrrooo_ooo'

>>> bwtViaSa("It_was_the_best_of_times_it_was_the_worst_of_times$")
's$esttssfftteww_hhmombootttt_i__woeeaaressIi_______'

>>> bwtViaSa('in_the_jingle_jangle_morning_Ill_come_following_you$')
'u_gleeengj_mlhl_nnnnt$nwj__lggIolo_iiiiarfcmlyo_o oo'

Python example: http://nbviewer.ipython.org/6798379
Burrows-Wheeler Transform

How to reverse the BWT?

BWM has a key property called the *LF Mapping*...
Burrows-Wheeler Transform: T-ranking

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

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

Now let’s re-write the BWM including ranks...
## Burrows-Wheeler Transform

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>BWM with T-ranking:</td>
<td>$ a_0 b_0 a_1 a_2 b_1 a_3$</td>
<td>$ a_3 a_0 b_0 a_1 a_2 b_1$</td>
</tr>
<tr>
<td></td>
<td>$ a_3 a_0 b_0 a_1 a_2 b_1$</td>
<td>$ a_1 a_2 b_1 a_3 $</td>
</tr>
<tr>
<td></td>
<td>$ a_2 b_1 a_3 $</td>
<td>$ a_0 b_0 a_1$</td>
</tr>
<tr>
<td></td>
<td>$ a_0 b_0 a_1 a_2 b_1 a_3 $</td>
<td>$ a_3 a_0 b_0 a_1 a_2$</td>
</tr>
<tr>
<td></td>
<td>$ b_1 a_3 $</td>
<td>$ a_0 b_0 a_1 a_2$</td>
</tr>
<tr>
<td></td>
<td>$ b_0 a_1 a_2 b_1 a_3 $</td>
<td>$ 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

**BWM with T-ranking:**

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

Same with **bs:**  b_1, b_0
Burrows-Wheeler Transform

Reversible permutation of the characters of a string, used originally for compression.

How is it useful for compression? How is it reversible? How is it an index?

Burrows-Wheeler Transform: LF Mapping

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

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 $T$

However we rank occurrences of $c$, ranks appear in the same order in $F$ and $L$
Burrows-Wheeler Transform: LF Mapping

Why does the LF Mapping hold?

They're sorted by right-context

Why are these as in this order relative to each other?

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

Whatever ranking we give to characters in T, rank orders in F and L will match
Burrows-Wheeler Transform: LF Mapping

BWM with T-ranking:

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

We’d like a different ranking so that for a given character, ranks are in ascending order as we look down the F / L columns...
Burrows-Wheeler Transform: LF Mapping

BWM with B-ranking:

\[
\begin{align*}
F & : \quad \$ \ a_3 \ b_1 \ a_1 \ a_2 \ b_0 \\
& \quad a_0 \ $ \ a_3 \ b_1 \ a_1 \ a_2 \ b_0 \\
& \quad a_1 \ a_2 \ b_0 \ a_3 \ $ \ a_3 \\
& \quad a_2 \ b_0 \ a_0 \ $ \ a_3 \ b_1 \ a_1 \\
& \quad a_3 \ b_1 \ a_1 \ a_2 \ b_0 \ a_0 \\
& \quad b_0 \ a_0 \ $ \ a_3 \ b_1 \ a_1 \\
& \quad b_1 \ a_1 \ a_2 \ b_0 \ a_0 \ $ \ a_3 \\
\end{align*}
\]

\[
\begin{align*}
L & : \quad a_0 \\
L & : \quad a_0 \\
L & : \quad b_0 \\
L & : \quad a_0 \\
L & : \quad a_3 \\
\end{align*}
\]

Ascending rank

\[ F \text{ now has very simple structure: a } \$, \text{ a block of } a \text{s with ascending ranks, a block of } b \text{s with ascending ranks}\]
Which BWM row begins with $b_1$?

Answer: row 6
Say $T$ has 300 $A$s, 400 $C$s, 250 $G$s and 700 $T$s and $\$ < A < C < G < T$

Which BWM row (0-based) begins with $G_{100}$? (Ranks are B-ranks.)

Skip row starting with $\$ $ (1 row)
Skip rows starting with $A$ (300 rows)
Skip rows starting with $C$ (400 rows)
Skip first 100 rows starting with $G$ (100 rows)

Answer: row $1 + 300 + 400 + 100 = \text{row 801}$
Burrows-Wheeler Transform: reversing

Reverse BWT(T) starting at right-hand-side of $T$ and moving left

**Start** in first row. $F$ must have $. $L$ contains character just prior to $:$ $a_0$

$a_0$: LF Mapping says this is same occurrence of $a$ as first $a$ in $F$. **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

Reverse of chars we visited = $a_3 \ b_1 \ a_1 \ a_2 \ b_0 \ a_0 \ $ = $T$
Burrows-Wheeler Transform: reversing

Another way to visualize reversing BWT(T):

\[ T: \ a_3 \ b_1 \ a_1 \ a_2 \ b_0 \ a_0 \ \$

Burrows-Wheeler Transform: reversing

def rankBwt(bw):
    ''' Given BWT string bw, return parallel list of B-ranks. Also
    returns tots: map from character to # times it appears. '''
    tots = dict()
    ranks = []
    for c in bw:
        if c not in tots: tots[c] = 0
        ranks.append(tots[c])
        tots[c] += 1
    return ranks, tots

def firstCol(tots):
    ''' Return map from character to the range of rows prefixed by
    the character. '''
    first = {}
    totc = 0
    for c, count in sorted(tots.iteritems()):
        first[c] = (totc, totc + count)
        totc += count
    return first

def reverseBwt(bw):
    ''' Make T from BWT(T) '''
    ranks, tots = rankBwt(bw)
    first = firstCol(tots)
    rowi = 0 # start in first row
    t = '$' # start with rightmost character
    while bw[rowi] != '$':
        c = bw[rowi]
        t = c + t # prepend to answer
        # jump to row that starts with c of same rank
        rowi = first[c][0] + ranks[rowi]
    return t

Calculate B-ranks and count occurrences of each char

Make concise representation of first BWM column

Do reversal

Python example: http://nbviewer.ipython.org/6860491
Burrows-Wheeler Transform: reversing

```python
def reverseBwt(bw):
    ''' Make T from BWT(T) '''
    ranks, tots = rankBwt(bw)
    first = firstCol(tots)
    rowi = 0  # start in first row
    t = '$'  # start with rightmost character
    while bw[rowi] != '$':
        c = bw[rowi]
        t = c + t  # prepend to answer
        # jump to row that starts with c of same rank
        rowi = first[c][0] + ranks[rowi]
    return t
```

ranks list is $m$ integers long! We’ll fix later.
Burrows-Wheeler Transform

We’ve seen how BWT is useful for compression:

Sorts characters by right-context, making a more compressible string

And how it’s reversible:

Repeated applications of LF Mapping, recreating \( T \) from right to left

How is it used as an index?
FM Index

FM Index: an index combining the BWT with a few small auxiliary data structures

“FM” supposedly stands for “Full-text Minute-space.” (But inventors are named Ferragina and Manzini)

Core of index consists of $F$ and $L$ from BWM:

$F$ can be represented very simply
(1 integer per alphabet character)

And $L$ is compressible

Potentially very space-economical!

**FM Index: querying**

Though BWM is related to suffix array, we can’t query it the same way.

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

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

We don’t have these columns; binary search isn’t possible.
FM Index: querying

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

Do this for $P$'s shortest suffix, then extend to successively longer suffixes until range becomes empty or we’ve exhausted $P$

\[
P = \text{aba}
\]

Easy to find all the rows beginning with $a$, thanks to $F$’s simple structure
FM Index: querying

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

\[ P = aba \]

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

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

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

Now we have the rows with prefix \(ba\)
FM Index: querying

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

\[ P = \textbf{aba} \]

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

\[ P = \textbf{aba} \]

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

Use LF Mapping

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

Now we have the rows with prefix \textbf{aba}
Now we have the same range, \([3, 5)\), we would have got from querying suffix array.

Unlike suffix array, we don’t immediately know where the matches are in T...
FM Index: querying

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

$P = \textbf{bba}$

Rows with \textbf{ba} prefix

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

No bs!
FM Index: querying

If we *scan* characters in the last column, that can be very slow, $O(m)$

$$P = \textbf{aba}$$

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

Scan, looking for *bs*
FM Index: lingering issues

(1) Scanning for preceding character is slow

\[ \begin{array}{cccccccccc}
$ & a & b & a & a & b & a_0 \\
 a_0 & $ & a & b & a & a & b_0 \\
a_1 & a & b & a & $ & a & b_1 \\
a_2 & b & a & $ & a & b & a_1 \\
a_3 & b & a & a & b & a & $ \\
b_0 & a & $ & a & b & a & a_2 \\
b_1 & a & a & b & a & $ & a_3 \\
\end{array} \]

\( O(m) \) scan

(2) Storing ranks takes too much space

```python
def reverseBwt(bw):
    """ Make T from BWT(T) ""
    ranks, tots = rankBwt(bw)
    first = firstCol(tots)
    rowi = 0
    t = "$"
    while bw[rowi] != '$':
        c = bw[rowi]
        t = c + t
        rowi = first[c][0] + ranks[rowi]
    return t
```

(3) Need way to find where matches occur in \( T \):

\[ \begin{array}{cccccccccc}
$ & a & b & a & a & b & a_0 \\
a_0 & $ & a & b & a & a & b_0 \\
a_1 & a & b & a & $ & a & b_1 \\
a_2 & b & a & $ & a & b & a_1 \\
a_3 & b & a & a & b & a & $ \\
b_0 & a & $ & a & b & a & a_2 \\
b_1 & a & a & b & a & $ & a_3 \\
\end{array} \]

Where?
Is there an $O(1)$ way to determine which $b$s precede the $a$s in our range?

---

**Idea:** pre-calculate $a$s, $b$s in $L$ up to every row:

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

We infer $b_0$ and $b_1$ appear in $L$ in this range $O(1)$ time, but requires $m \times |\sum| \text{ integers}$

---

$\text{Occ}(c, p) = \# \text{ of of } c \text{ in the first } p \text{ characters of BWT}(S)$, aka the LF mapping.

---

— also referred to as $\text{Occ}(c, k)$
FM Index: fast rank calculations

Is there an $O(1)$ way to determine which $b$s precede the $a$s in our range?

Idea: pre-calculate # $a$s, $b$s in $L$ up to every row:

<table>
<thead>
<tr>
<th>$F$</th>
<th>$L$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$</td>
<td>$a$</td>
</tr>
<tr>
<td>$a$</td>
<td>$b$</td>
</tr>
<tr>
<td>$a$</td>
<td>$a$</td>
</tr>
<tr>
<td>$b$</td>
<td>$a$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$F$</th>
<th>$L$</th>
<th>Tally --- also referred to as $\text{Occ}(c, k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$</td>
<td>$a$</td>
<td>1 0</td>
</tr>
<tr>
<td>$a$</td>
<td>$a$</td>
<td>1 1</td>
</tr>
<tr>
<td>$a$</td>
<td>$b$</td>
<td>1 2</td>
</tr>
<tr>
<td>$a$</td>
<td>$a$</td>
<td>2 2</td>
</tr>
<tr>
<td>$a$</td>
<td>$$</td>
<td>2 2</td>
</tr>
<tr>
<td>$b$</td>
<td>$a$</td>
<td>3 2</td>
</tr>
<tr>
<td>$b$</td>
<td>$a$</td>
<td>4 2</td>
</tr>
</tbody>
</table>

$\text{Occ}(c, k) = \# \text{ of } c \text{ in the first } k \text{ characters of BWT(S), aka the LF mapping.}$

$O(1)$ time, but requires $m \times |\Sigma|$ integers

We infer $b_0$ and $b_1$ appear in $L$ in this range.
Another idea: pre-calculate \# as, bs in $L$ up to some rows, e.g. every 5\textsuperscript{th} row. Call pre-calculated rows checkpoints.

<table>
<thead>
<tr>
<th>F</th>
<th>L</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>a</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lookup here succeeds as usual

Oops: not a checkpoint

But there’s one nearby

To resolve a lookup for character $c$ in non-checkpoint row, scan along $L$ until we get to nearest checkpoint. Use tally at the checkpoint, adjusted for \# of cs we saw along the way.
FM Index: fast rank calculations

Assuming checkpoints are spaced $O(1)$ distance apart, lookups are $O(1)$

What's my rank?

$482 + 2 - 1 = 483$

What's my rank?

$439 - 2 - 1 = 436$
FM Index: a few problems

Solved! At the expense of adding checkpoints \(O(m)\) integers to index.

(1) This scan is \(O(m)\) work

\[
\begin{array}{cccccc}
\$ & a & b & a & a & b \\
a_0 & $ & a & b & a & a \\
a_1 & a & b & a & $ & a \\
a_2 & b & a & $ & a & b \\
a_3 & b & a & a & b & \$
\end{array}
\]

With checkpoints it’s \(O(1)\)

(2) Ranking takes too much space

\[
\begin{array}{cccccc}
\$ & a & b & a & a & b \\
a_0 & $ & a & b & a & a \\
a_1 & a & b & a & $ & a \\
a_2 & b & a & $ & a & b \\
a_3 & b & a & a & b & \$
\end{array}
\]

With checkpoints, we greatly reduce # integers needed for ranks

But it’s still \(O(m)\) space - there’s literature on how to improve this space bound

```python
def reverseBwt(bw):
    """ Make T from BWT(T) """
    ranks, tots = rankBwt(bw)
    first = firstCol(tots)
    rowi = 0
    t = "$"
    while bw[rowi] != '$':
        c = bw[rowi]
        t = c + t
        rowi = first[c][0] + ranks[rowi]
    return t
```
FM Index: a few problems

Not yet solved: (3) Need a way to find where these occurrences are in $T$:

$\begin{align*}
a & \quad a \quad a \quad a \quad b \\
a_0 & \quad a \quad b \quad a \quad a \quad b_0 \\
a_1 & \quad a \quad b \quad a \quad b \quad a_1 \\
a_2 & \quad b \quad a \quad a \quad b \quad a_1 \\
a_3 & \quad b \quad a \quad a \quad a \quad b \\
b_0 & \quad a \quad b \quad a \quad a \quad a_2 \\
b_1 & \quad a \quad b \quad a \quad a \quad b_3
\end{align*}$

If suffix array were part of index, we could simply look up the offsets

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

Offsets: 0, 3

But SA requires $m$ integers
FM Index: resolving offsets

Idea: store some, but not all, entries of the suffix array

<table>
<thead>
<tr>
<th>F</th>
<th>L</th>
<th>SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a b a a b a</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>a $ a b a a b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a a b a $ a b</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>a b a $ a b a</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>a b a a b a $</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>b a $ a b a a</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>b a a b a $ a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lookup for row 4 succeeds - we kept that entry of SA
Lookup for row 3 fails - we discarded that entry of SA
**FM Index: resolving offsets**

But LF Mapping tells us that the \textbf{a} at the end of row 3 corresponds to...

...the \textbf{a} at the beginning of row 2

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

SA

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

And row 2 has a suffix array value = 2

So row 3 has suffix array value = 3 = 2 (row 2’s SA val) + 1 (# steps to row 2)

If saved SA values are $O(1)$ positions apart in $T$, resolving offset is $O(1)$ time
At the expense of adding some SA values ($O(m)$ integers) to index

Call this the “SA sample”

Need a way to find where these occurrences are in $T$:

```
$ a b a a b a_0
a_0 $ a b a a b_0
a_1 a b a $ a b_1
a_2 b a $ a b a_1
a_3 b a a b a $

b_0 a $ a b a a_2
b_1 a a b a $ a_3
```

With SA sample we can do this in $O(1)$ time per occurrence
FM Index: small memory footprint

Components of the FM Index:

First column ($F$): $\sim |\Sigma|$ integers

Last column ($L$): $m$ characters

SA sample: $m \cdot a$ integers, where $a$ is fraction of rows kept

Checkpoints: $m \times |\Sigma| \cdot b$ integers, where $b$ is fraction of rows checkpointed

Example: DNA alphabet (2 bits per nucleotide), $T =$ human genome, $a = 1/32$, $b = 1/128$

First column ($F$): 16 bytes

Last column ($L$): 2 bits $\times$ 3 billion chars $= 750$ MB

SA sample: 3 billion chars $\times$ 4 bytes/char / 32 $= \sim 400$ MB

Checkpoints: 3 billion $\times$ 4 bytes/char / 128 $= \sim 100$ MB

Total $< 1.5$ GB
Actual FM-Index Built on Compressed String


Data structure has “space occupancy that is a function of the entropy of the underlying data set”

Stores text $T[1,u]$ in $O(H_k(T)) + o(1)$ bits for $k \geq 0$ where $H_k(T)$ is the kth order empirical entropy of the text — **sub-linear** for a compressible string

---

**Theorem 1** Let $Z$ denote the output of the algorithm BW.RLX on input $T[1,u]$. The number of occurrences of a pattern $P[1,p]$ in $T[1,u]$ can be computed in $O(p)$ time on a RAM. The space occupancy is $|Z| + O\left(\frac{u}{\log u} \log \log u\right)$ bits in the worst case.

---

**Theorem 2** A text $T[1,u]$ can be preprocessed in $O(u)$ time so that all the occ occurrences of a pattern $P[1,p]$ in $T$ can be listed in $O(p + occ \log^2 u)$ time on a RAM. The space occupancy is bounded by $5H_k(T) + O\left(\frac{\log \log u}{\log u}\right)$ bits per input symbol in the worst case, for any fixed $k \geq 0$.

---

**Theorem 3** A text $T[1,u]$ can be indexed so that all the occ occurrences of a pattern $P[1,p]$ in $T$ can be listed in $O(p + occ \log^4 u)$ time on a RAM. The space occupancy is $O(H_k(T) + \frac{\log \log u}{\log^2 u})$ bits per input symbol in the worst case, for any fixed $k \geq 0$. 

Computing BWT in $O(n)$ time

- Easy $O(n^2 \log n)$-time algorithm to compute the BWT (create and sort the BWT matrix explicitly).

- Several direct $O(n)$-time algorithms for BWT. These are space efficient. (Bowtie e.g. uses [1])

- Also can use suffix arrays or trees:
  - Compute the suffix array, use correspondence between suffix array and BWT to output the BWT.
  - $O(n)$-time and $O(n)$-space, but the constants are large.

Compressing BWT Strings

Lots of possible compression schemes will benefit from preprocessing with BWT (since it tends to group runs of the same letters together).

One good scheme proposed by Ferragina & Manzini:

\[
\text{PrefixCode}(\text{rle}(\text{MTF}(\text{BWT}(S))))
\]

*slide courtesy of Carl Kingsford*
Move-To-Front Coding

To encode a letter, use its index in the current list, and then move it to the front of the list.

\[
\sum \text{ do$oodwg}
\]

List with all letters from the allowed alphabet

- $dgow \quad 1$
- d$gow \quad 13$
- od$gw \quad 132$
- $odgw \quad 1321$
- o$dgw \quad 13210$
- o$dgw \quad 132102$
- do$gw \quad 1321024$
- wdo$g \quad 13210244 = \text{MTF(do$oodwg)}$

Benefits:
- Runs of the same letter will lead to runs of 0s.
- Common letters get small numbers, while rare letters get big numbers.

*slide courtesy of Carl Kingsford*
Computing Occ in Compressed String

Break BWT(S) into blocks of length L (we will decide on a value for L later):

```
  BWT(S)
  BT_1  BT_2  BT_3  ...  __________  __________  __________  __________  __________

  PrefixCode(rle(MTF(BWT(BT_2))))

  Occ(c, p) = # of “c” up thru p

  BZ_1  BZ_2  BZ_3  ...  __________  __________  __________  __________  __________
```

Assumes every run of 0s is contained in a block [just for ease of explanation].
We will store some extra info for each block (and some groups of blocks) to compute Occ(c, p) quickly.
Extra Info to Compute Occ

block: store $|\Sigma|$-long array giving # of occurrences of each character up thru and including this block since the end of the last super block.

superblock: store $|\Sigma|$-long array giving # of occurrences of each character up thru and including this superblock

*slide courtesy of Carl Kingsford*
Extra Info to Compute Occ

\( u = \text{compressed length} \)

Choose \( L = O(\log u) \)

\( \frac{u}{L} \) blocks, each array is \(|\Sigma| \log L\) space

\[ \frac{u}{L} \log L = \frac{u}{\log u} \log \log u \] total space.

\( \text{block: store } |\Sigma| \)-long array giving # of occurrences of each character up thru and including this block since the end of the last super block.

\( \text{superblock: store } |\Sigma| \)-long array giving # of occurrences of each character up thru and including this superblock

*slide courtesy of Carl Kingsford*
**Extra Info to Compute Occ**

\( u = \text{compressed length} \)

Choose \( L = O(\log u) \)

\( u/L \) blocks, each array is \(|\Sigma|\log L\) space

\[
\frac{u}{L} \log L = \frac{u}{\log u} \log \log u \quad \text{total space.}
\]

**block:** store \(|\Sigma|\)-long array giving # of occurrences of each character up thru and including this block since the end of the last super block.

![Diagram of superblock and block]

**superblock:** store \(|\Sigma|\)-long array giving # of occurrences of each character up thru and including this superblock

\( u/L^2 \) superblocks, each array is \(|\Sigma|\log u\) long

\[
\Rightarrow \quad \frac{u}{(\log u)^2} \log u = \frac{u}{\log u} \quad \text{total space.}
\]

*slide courtesy of Carl Kingsford*
Extra Info to Compute Occ

\[ u = \text{compressed length} \]

Choose \( L = O(\log u) \)

\[ \text{block} \]

\[ \text{superblock} \]

\[ \text{Occ}(c, p) = \# \text{ of } \text{c} \text{ up thru } p: \]

sum value at last superblock, value at end of previous block, but then need to handle this block.

Store an array: \( M[c, k, BZ_i, MTF_i] = \# \text{ of occurrences of c through the kth letter of a block of type (BZ}_i, MTF_i). \)

Size: \( O(|\Sigma|L^2|\Sigma|) = O(L^2^L) = O(u^c \log u) \) for \( c < 1 \) (since the string is compressed)

*slide courtesy of Carl Kingsford