Genome Indexing and the Burrows-Wheeler Transform

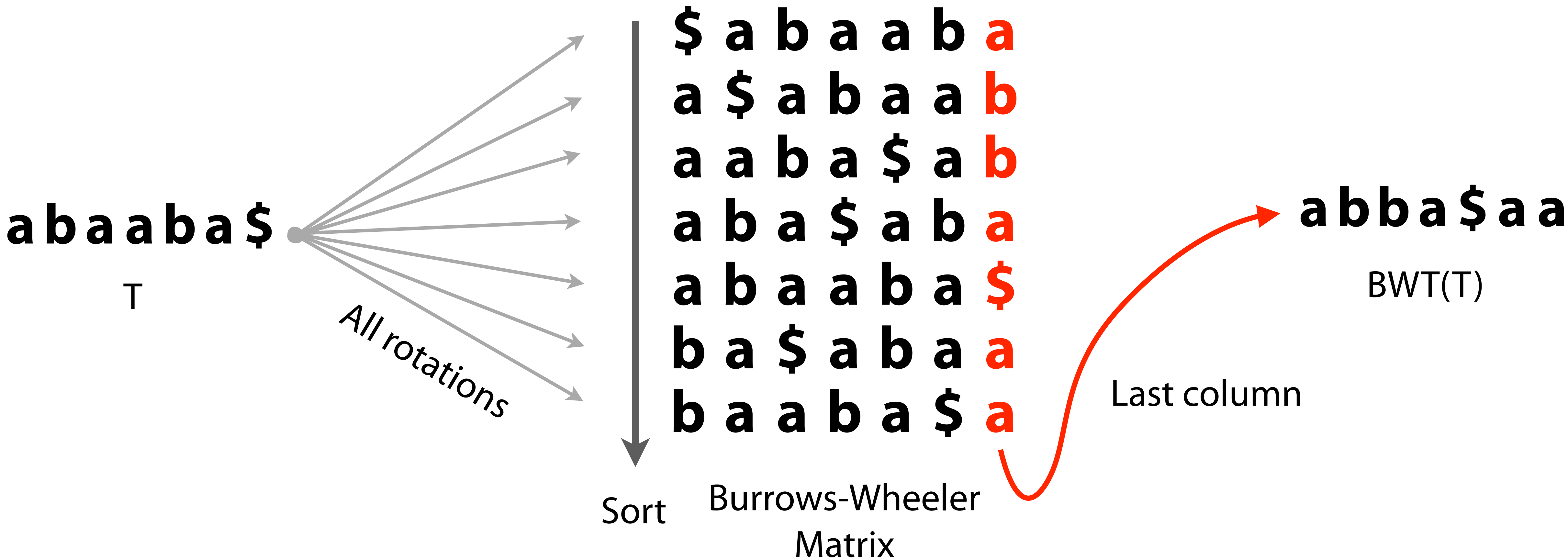
Alan Kuhnle

25 October 2018
StringBio 2018
Slides adapted from Ben Langmead, Travis Gagie

UF UNIVERSITY of FLORIDA

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 M, Wheeler DJ: A block sorting lossless data compression algorithm. *Digital Equipment Corporation, Palo Alto, CA* 1994, Technical Report 124; 1994

Burrows-Wheeler Transform

```
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)))
```

Make list of all rotations

Sort them

Take last column

```
>>> bwtViaBwm("Tomorrow_and_tomorrow_and_tomorrow$")
'w$wwdd__nnoooaattTmmmrrrrrrooo__ooo'

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

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

Python example: 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

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

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 M, Wheeler DJ: A block sorting lossless data compression algorithm. *Digital Equipment Corporation, Palo Alto, CA* 1994, Technical Report 124; 1994

Burrows-Wheeler Transform

BWM bears a resemblance to the suffix array

```
$ a b a a b a
a $ a b a a b
a a b a $ a b
a b a $ a b a
a b a a b a $
b a $ a b a a
b a a b a $ a
```

BWM(T)

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

SA(T)

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"

BWM(T)		SA(T)
$ a b a a b a	6	$
a $ a b a a b	5	a $
a a b a $ a b	2	a a b a $
a b a $ a b a	3	a b a $
a b a a b a $	0	a b a a b a $
b a $ a b a a	4	b a $
b a a b a $ a	1	b a a b a $

Burrows-Wheeler Transform

```
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)
```

Make suffix array

```
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
```

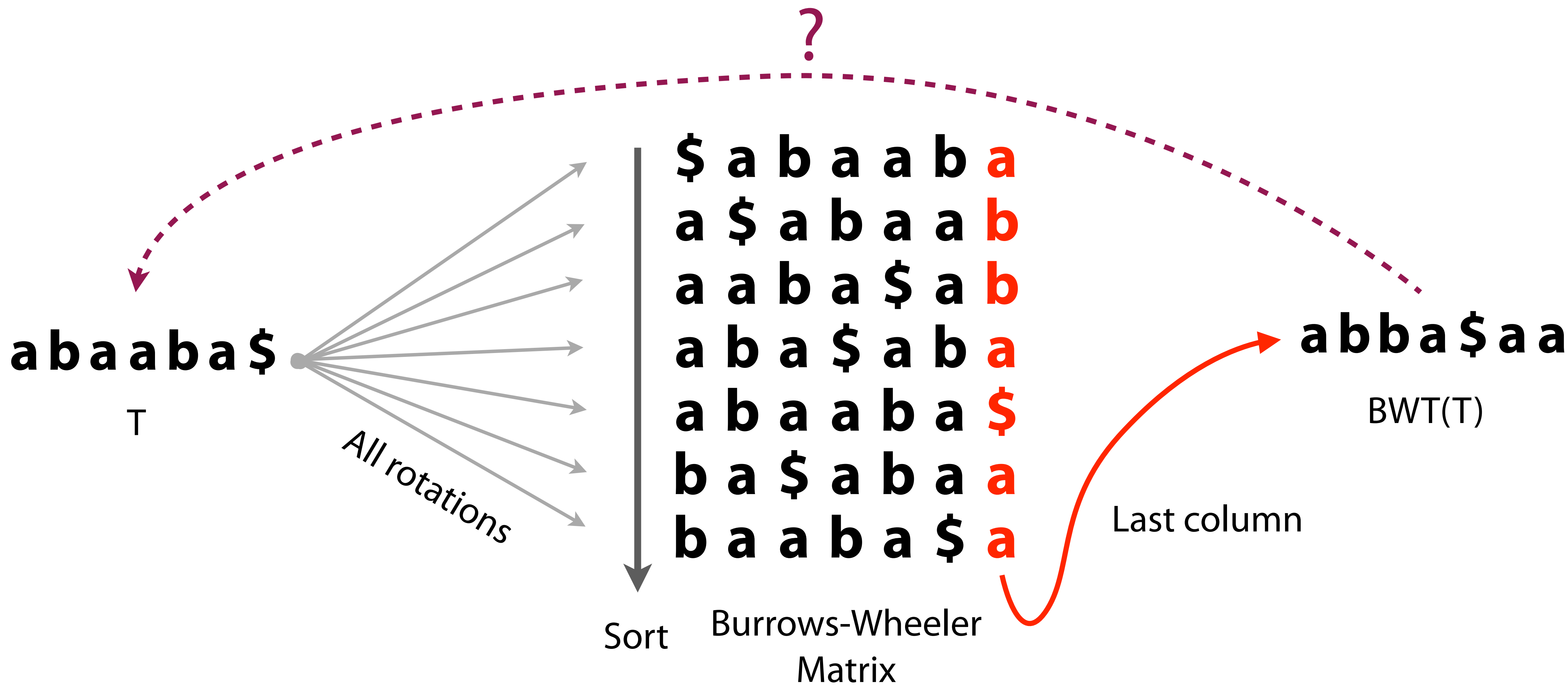
Take characters just to the left of the sorted suffixes

```
>>> bwtViaSa("Tomorrow_and_tomorrow_and_tomorrow$")
'w$wwdd__nnoooaattTmmmrrrrrrooo__ooo'

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

>>> bwtViaSa('in_the_jingle_jangle_morning_Ill_come_following_you$')
'u_gleeeengj_mlhl_nnnnt$nwj__lggIolo_iiiiarfcmylo_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*.

$$\mathbf{a}_0\ \mathbf{b}_0\ \mathbf{a}_1\ \mathbf{a}_2\ \mathbf{b}_1\ \mathbf{a}_3\ \$$$

Now let's re-write the BWM including ranks...

Burrows-Wheeler Transform

BWM with T-ranking:

F						*L*
\$	a_0	b_0	a_1	a_2	b_1	$\mathbf{a_3}$
$\mathbf{a_3}$	\$	a_0	b_0	a_1	a_2	b_1
$\mathbf{a_1}$	a_2	b_1	a_3	\$	a_0	b_0
$\mathbf{a_2}$	b_1	a_3	\$	a_0	b_0	$\mathbf{a_1}$
$\mathbf{a_0}$	b_0	a_1	a_2	b_1	a_3	\$
b_1	a_3	\$	a_0	b_0	a_1	$\mathbf{a_2}$
b_0	a_1	a_2	b_1	a_3	\$	$\mathbf{a_0}$

Look at first and last columns, called *F* and *L*

And look at just the **a**s

as occur in the same order in *F* and *L*. As we look down columns, in both cases we see: $\mathbf{a_3}$, $\mathbf{a_1}$, $\mathbf{a_2}$, $\mathbf{a_0}$

Burrows-Wheeler Transform

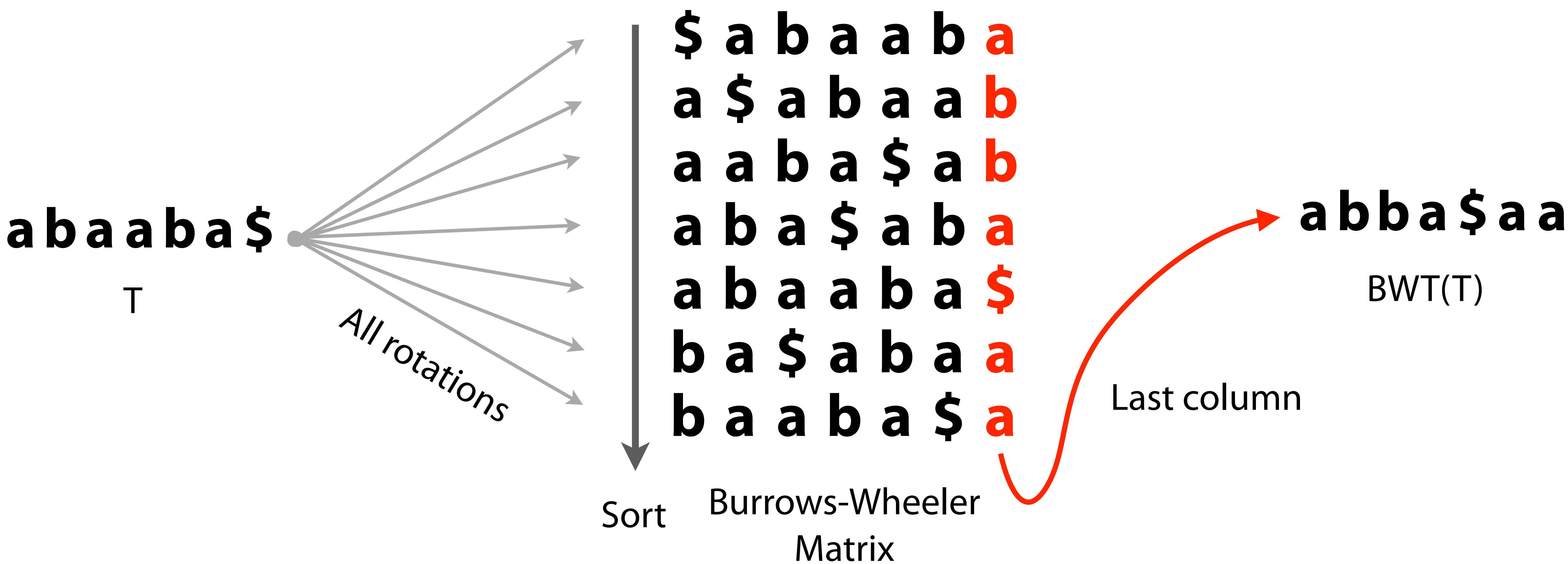
BWM with T-ranking:

F						*L*
$\$$	a_0	b_0	a_1	a_2	b_1	a_3
a_3	$\$$	a_0	b_0	a_1	a_2	$\mathbf{b}_1$
a_1	a_2	b_1	a_3	$\$$	a_0	$\mathbf{b}_0$
a_2	b_1	a_3	$\$$	a_0	b_0	a_1
a_0	b_0	a_1	a_2	b_1	a_3	$\$$
$\mathbf{b}_1$	a_3	$\$$	a_0	b_0	a_1	a_2
$\mathbf{b}_0$	a_1	a_2	b_1	a_3	$\$$	a_0

Same with **b**s: $\mathbf{b}_1$, $\mathbf{b}_0$

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 M, Wheeler DJ: A block sorting lossless data compression algorithm. *Digital Equipment Corporation, Palo Alto, CA* 1994, Technical Report 124; 1994

Burrows-Wheeler Transform: LF Mapping

BWM with T-ranking:

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

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?

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

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

They're sorted by right-context

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

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

They're sorted by right-context

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:

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

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:

F						*L*
$	a_3	b_1	a_1	a_2	b_0	a_0
a_0	$	a_3	b_1	a_1	a_2	b_0
a_1	a_2	b_0	a_3	$	a_3	b_1
a_2	b_0	a_0	$	a_3	b_1	a_1
a_3	b_1	a_1	a_2	b_0	a_0	$
b_0	a_0	$	a_3	b_1	a_1	a_2
b_1	a_1	a_2	b_0	a_0	$	a_3

Ascending rank

F now has very simple structure: a **$**, a block of **a**s *with ascending ranks*, a block of **b**s *with ascending ranks*

Burrows-Wheeler Transform

	F	*L*
	$	$\mathbf{a}_0$
	$\mathbf{a}_0$	$\mathbf{b}_0$
	$\mathbf{a}_1$	$\mathbf{b}_1$ ←
	$\mathbf{a}_2$	$\mathbf{a}_1$
	$\mathbf{a}_3$	**$**
	$\mathbf{b}_0$	$\mathbf{a}_2$
row 6 →	$\mathbf{b}_1$	$\mathbf{a}_3$

Which BWM row *begins* with $\mathbf{b}_1$?

Skip row starting with **$** (1 row)

Skip rows starting with **a** (4 rows)

Skip row starting with $\mathbf{b}_0$ (1 row)

Answer: row 6

Burrows-Wheeler Transform

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 $\mathbf{G}_{\mathbf{100}}$? (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 = **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 **$**. *L* contains character just prior to **$**: $\mathbf{a}_0$

$\mathbf{a}_0$: LF Mapping says this is same occurrence of **a** as first **a** in *F*. Jump to row *beginning* with $\mathbf{a}_0$. *L* contains character just prior to $\mathbf{a}_0$: $\mathbf{b}_0$.

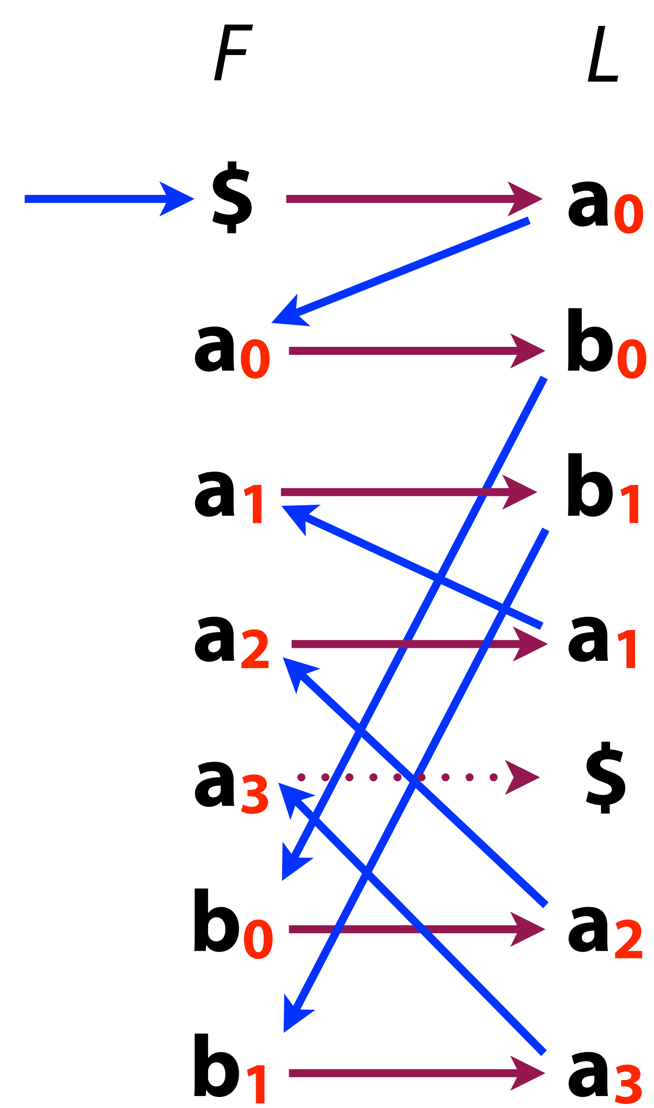
Repeat for $\mathbf{b}_0$, get $\mathbf{a}_2$

Repeat for $\mathbf{a}_2$, get $\mathbf{a}_1$

Repeat for $\mathbf{a}_1$, get $\mathbf{b}_1$

Repeat for $\mathbf{b}_1$, get $\mathbf{a}_3$

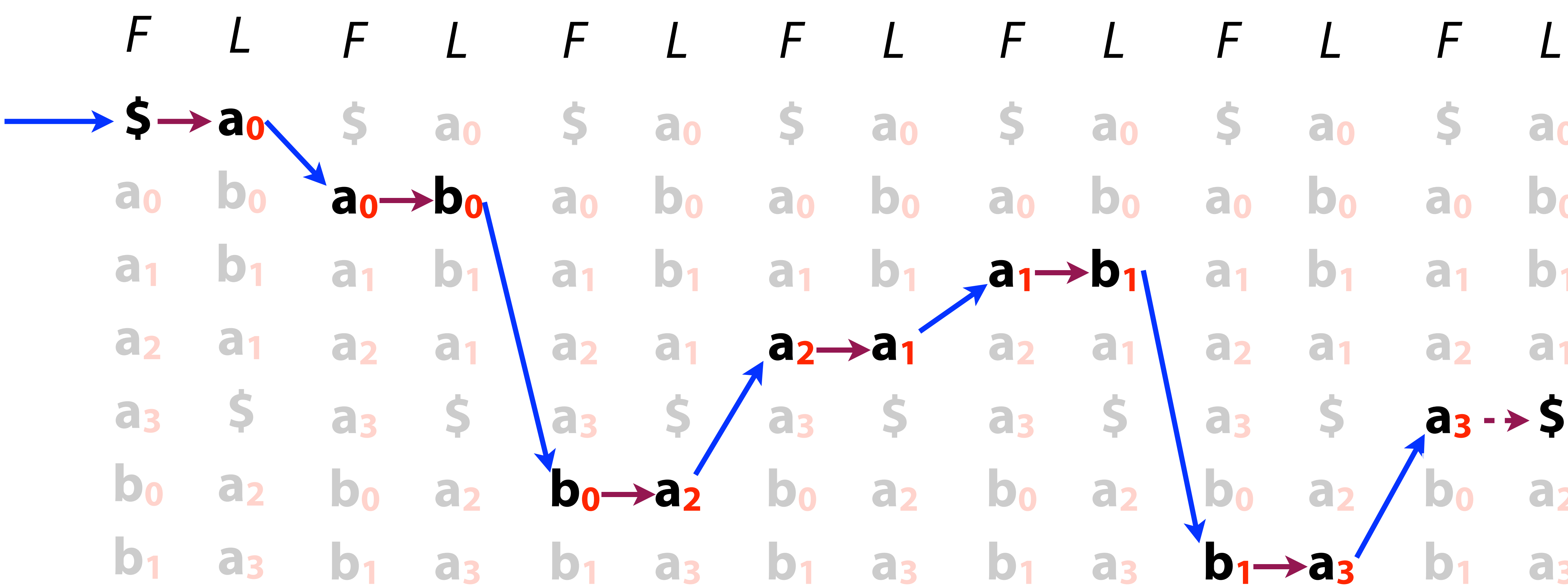
Repeat for $\mathbf{a}_3$, get **$**, done



Reverse of chars we visited = $\mathbf{a}_3\ \mathbf{b}_1\ \mathbf{a}_1\ \mathbf{a}_2\ \mathbf{b}_0\ \mathbf{a}_0\ \$ = T$

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
```

Calculate B-ranks and count occurrences of each char

```
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
```

Make concise representation of first BWM column

```
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
```

Do reversal

Python example: http://nbviewer.ipython.org/6860491

Burrows-Wheeler Transform: reversing

```
>>> reverseBwt("w$wwdd__nnoooaattTmmmrrrrrrooo__ooo")
'Tomorrow_and_tomorrow_and_tomorrow$'

>>> reverseBwt("s$esttssfftteww_hhmmbootttt_ii__woeeaaressIi_______")
'It_was_the_best_of_times_it_was_the_worst_of_times$'

>>> reverseBwt("u_gleeeengj_mlhl_nnnnt$nwj__lggIolo_iiiiarfcmylo_oo_")
'in_the_jingle_jangle_morning_Ill_come_following_you$'
```

ranks list is *m* integers long! We'll fix later.

```
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
```

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 auxilliary 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						*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**

Not stored in index

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

And *L* is compressible

Potentially very space-economical!

Paolo Ferragina, and Giovanni Manzini. "Opportunistic data structures with applications." *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE, 2000.

FM Index: querying

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

$	a	b	a	a	b	a
a	$	a	b	a	a	b
a	a	b	a	$	a	b
a	b	a	$	a	b	a
a	b	a	a	b	a	$
b	a	$	a	b	a	a
b	a	a	b	a	$	a

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 $

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 = **aba**

Easy to find all the rows beginning with **a**, thanks to *F*'s simple structure

F						*L*
$	a	b	a	a	b	$\mathbf{a}_3$
$\mathbf{a}_0$	$	a	b	a	a	$\mathbf{b}_1$
$\mathbf{a}_1$	a	b	a	$	a	$\mathbf{b}_0$
$\mathbf{a}_2$	b	a	$	a	b	$\mathbf{a}_1$
$\mathbf{a}_3$	b	a	a	b	a	**$**
$\mathbf{b}_0$	a	$	a	b	a	$\mathbf{a}_2$
$\mathbf{b}_1$	a	a	b	a	$	$\mathbf{a}_0$

FM Index: querying

We have rows beginning with **a**, now we seek rows beginning with **ba**

$P =$ **aba**

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	\$
b_0	a	\$	a	b	a	a_2
b_1	a	a	b	a	\$	a_3

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

$P =$ **aba**

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	\$
b_0	a	\$	a	b	a	a_2
b_1	a	a	b	a	\$	a_3

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 **ba**, now we seek rows beginning with **aba**

$P =$ **aba**

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	\$
b_0	a	\$	a	b	a	a_2
b_1	a	a	b	a	\$	a_3

a_2, **a_3** occur just to left.

$P =$ **aba**

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	\$
b_0	a	\$	a	b	a	a_2
b_1	a	a	b	a	\$	a_3

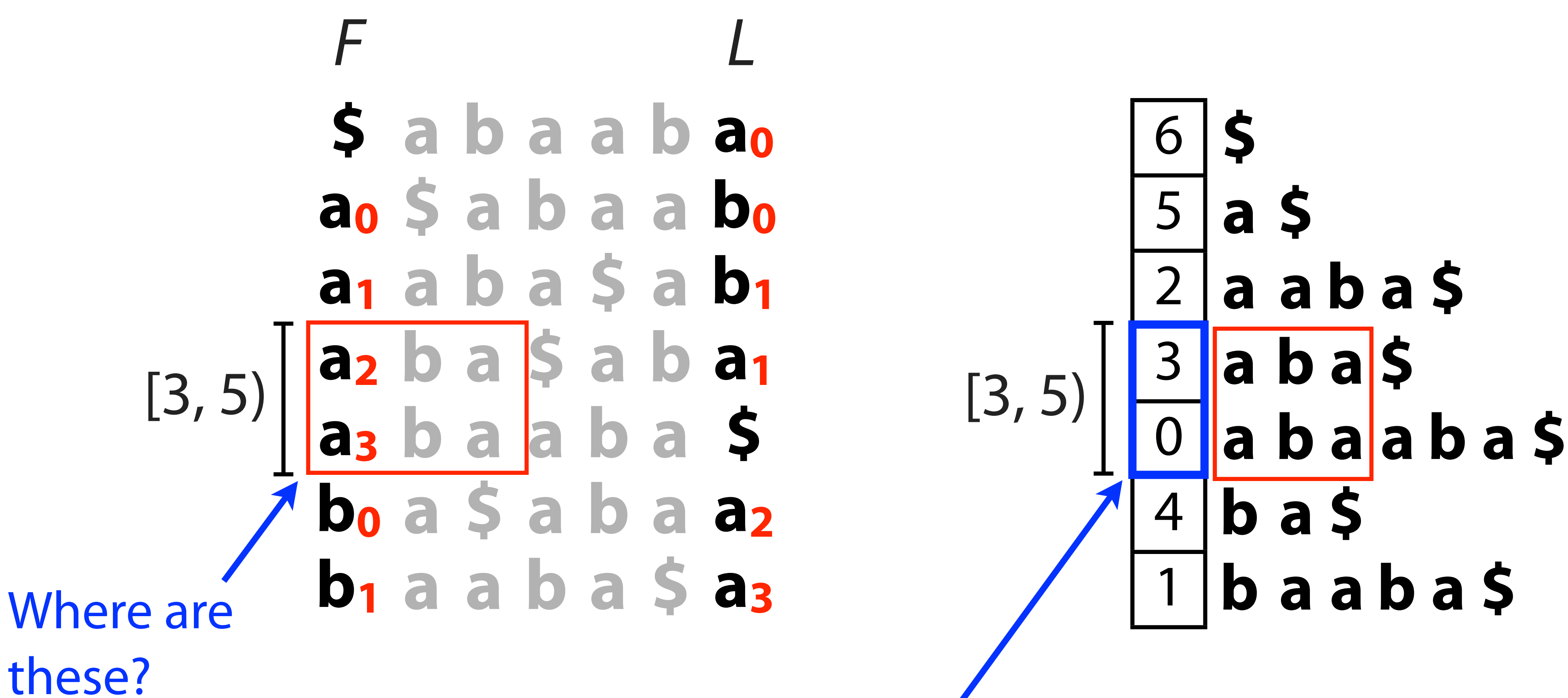
Use LF Mapping

Now we have the rows with prefix **aba**

FM Index: querying

P = **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*:

$P =$ **bba**

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

Rows with **ba** prefix

No **b**s!

FM Index: querying

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

$P =$ **aba**

F						L
$	a	b	a	a	b	a_3
a_0	$	a	b	a	a	b_1
a_1	a	b	a	$	a	b_0
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_0

Scan, looking for **b**s

FM Index: lingering issues

(1) Scanning for preceding character is slow

\$	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

$O(m)$ scan

(2) Storing ranks takes too much space

m integers

```
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*:

\$	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

Where?

FM Index: fast rank calculations

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

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

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

Tally

F	*L*	**a**	**b**
$	a	1	0
a	b	1	1
a	b	1	2
a	a	2	2
a	$	2	2
b	a	3	2
b	a	4	2

We infer $\mathbf{b}_0$ and $\mathbf{b}_1$ appear in *L* in this range

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

FM Index: fast rank calculations

Another idea: pre-calculate # **a**s, **b**s in *L* up to *some* rows, e.g. every 5th row. Call pre-calculated rows *checkpoints*.

F	*L*	*Tally* **a**	*Tally* **b**	
$	**a**	1	0	Lookup here succeeds as usual
a	**b**			
a	**b**			
a	**a**			
a	**$**			Oops: not a checkpoint
b	**a**	3	2	But there's one nearby
b	**a**			

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

What's my rank?

482 + 2 - 1 = 483

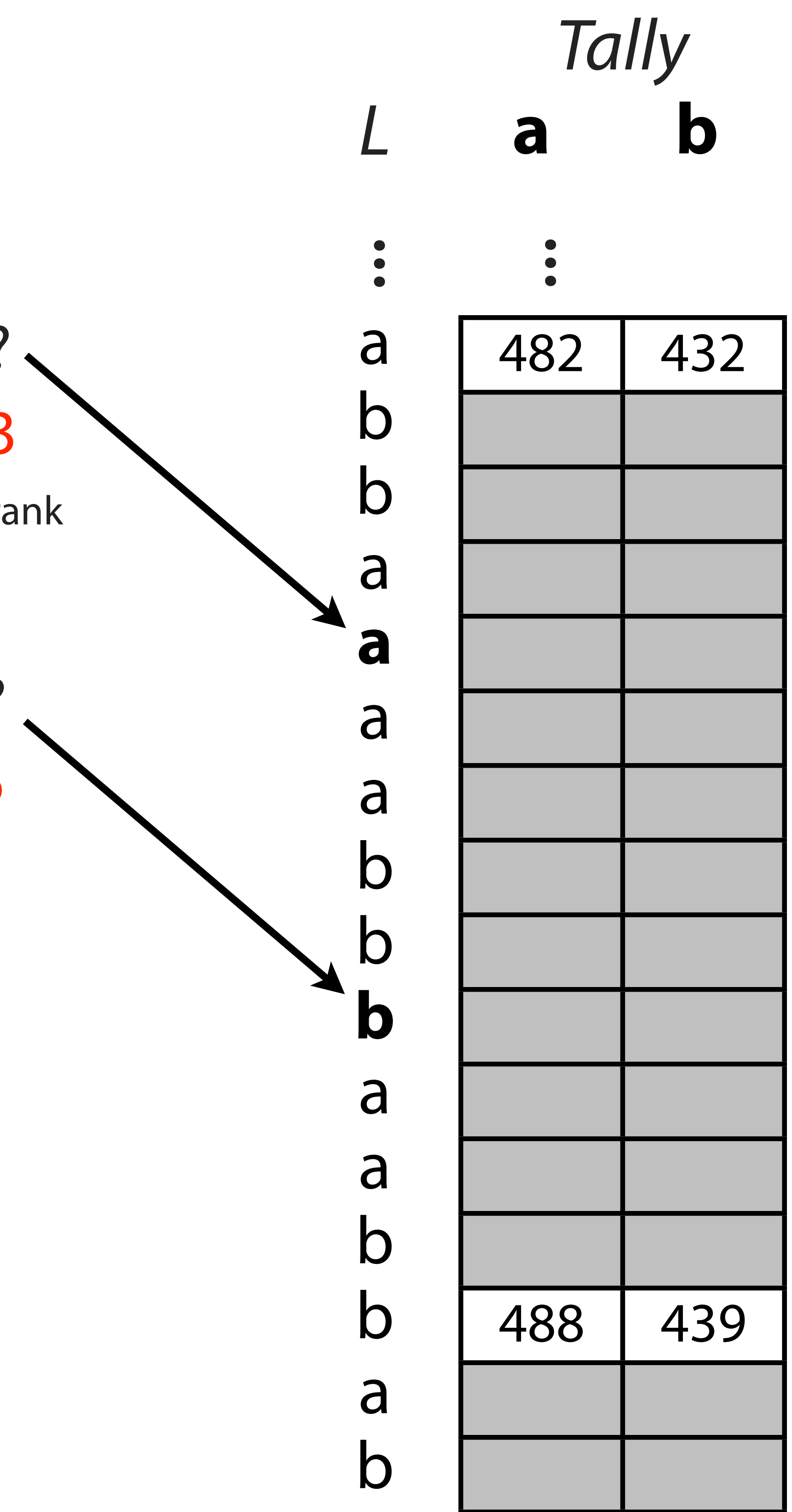
Checkpoint

as along the way

tally -> rank

What's my rank?

439 - 2 - 1 = 436

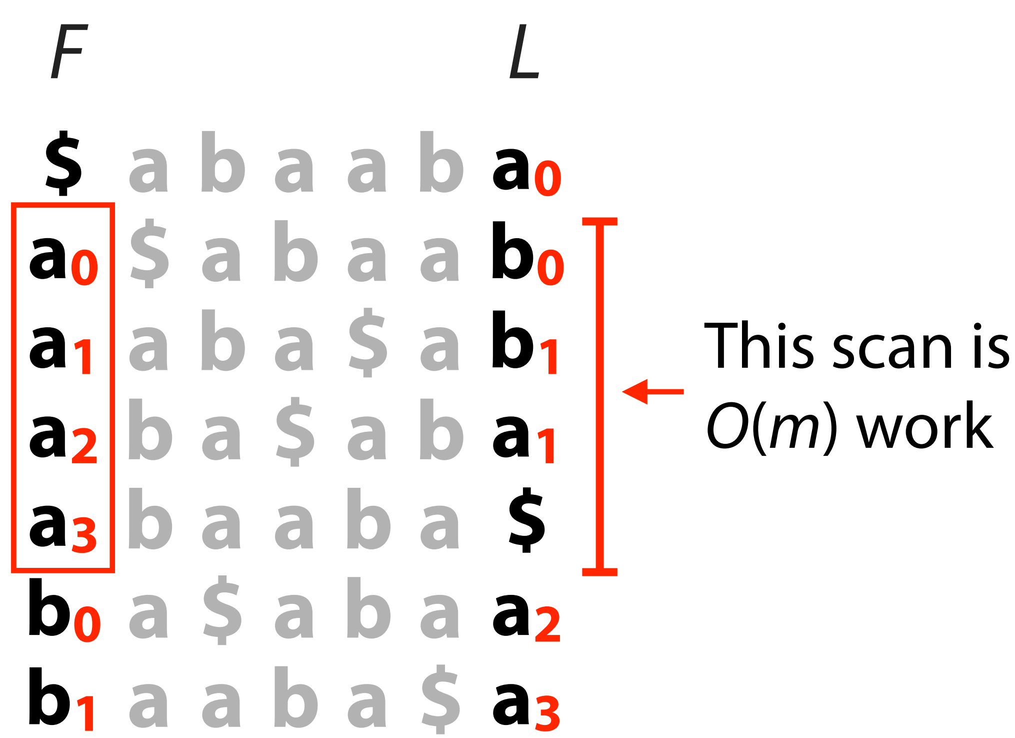


Assuming checkpoints are spaced $O(1)$ distance apart, lookups are $O(1)$

FM Index: a few problems

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

(1)



With checkpoints it's $O(1)$

(2) Ranking takes too much space

m integers

```
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
```

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

FM Index: a few problems

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

$	a	b	a	a	b	$\mathbf{a}_0$
$\mathbf{a}_0$	$	a	b	a	a	$\mathbf{b}_0$
$\mathbf{a}_1$	a	b	a	$	a	$\mathbf{b}_1$
$\mathbf{a}_2$	b	a	$	a	b	$\mathbf{a}_1$
$\mathbf{a}_3$	b	a	a	b	a	**$**
$\mathbf{b}_0$	a	$	a	b	a	$\mathbf{a}_2$
$\mathbf{b}_1$	a	a	b	a	$	$\mathbf{a}_3$

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

F						*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**

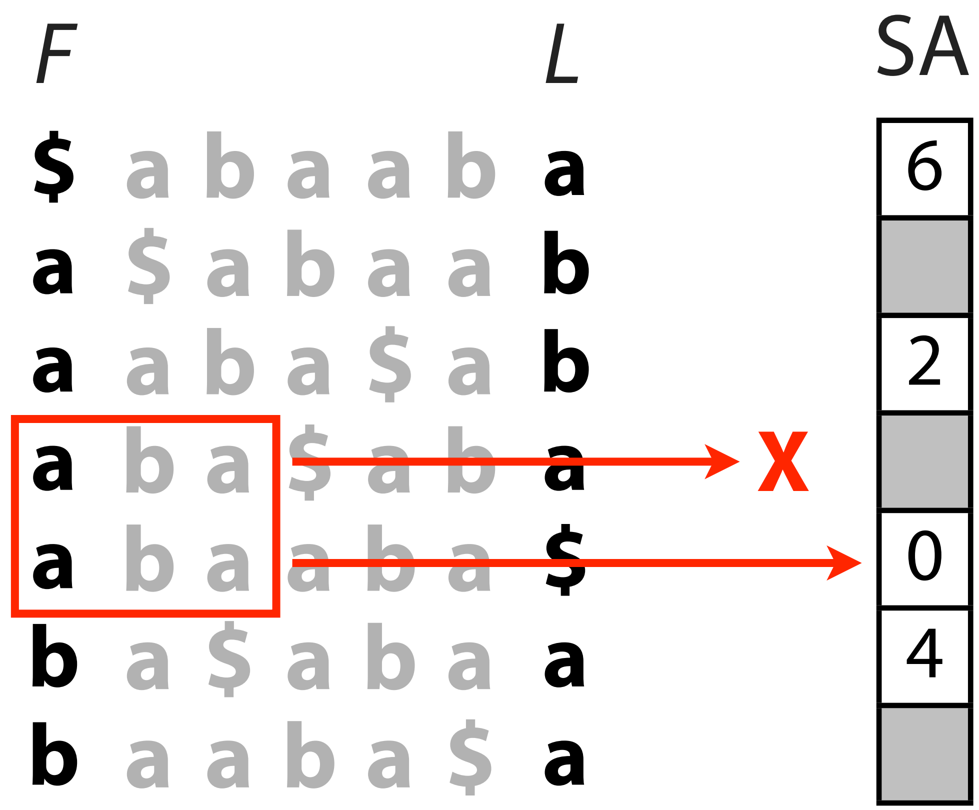
Offsets: 0, 3

SA	
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 $**

But SA requires *m* integers

FM Index: resolving offsets

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



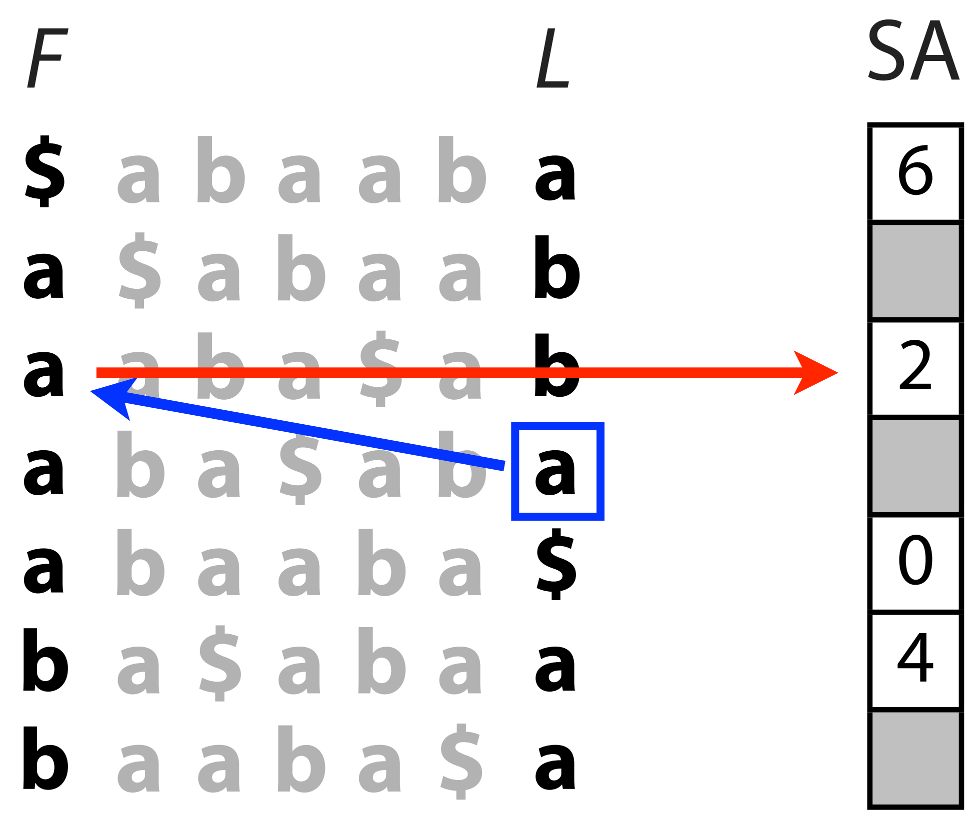
Lookup for row 4 succeeds - we kept that entry of SA

Lookup for row 3 fails - we discarded that entry of SA

FM Index: resolving offsets

But LF Mapping tells us that the **a** at the end of row 3 corresponds to...

...the **a** at the begining of row 2



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

FM Index: problems solved

Solved!

At the expense of adding some SA values ($O(m)$ integers) to index

Call this the "SA sample"

(3) 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):	~ $\lvert\Sigma\rvert$ integers
Last column (L):	m characters
SA sample:	$m \cdot a$ integers, where a is fraction of rows kept
Checkpoints:	$m \times \lvert\Sigma\rvert \cdot b$ 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 * 3 billion chars = 750 MB
SA sample:	3 billion chars * 4 bytes/char / 32 = ~ 400 MB
Checkpoints:	*3 billion* * 4 bytes/char / *128* = ~ 100 MB

Total < 1.5 GB

NGBWT:
BWT Tools for NGS Datasets

Travis Gagie
et al.

University of Pisa
July 16th, 2018

abstract

The Burrows-Wheeler Transform (BWT) is the basis of several tools that have enabled the genomics revolution, but researchers developing those tools are now victims of their own success: next-generation sequencing (NGS) has resulted in genomic databases so large that we strain to build their BWTs, and we can no longer afford the auxiliary data structures we used to take for granted. We are now losing functionality in practice because some of our techniques do not scale. The BWTs themselves remain small and fast and beautiful, however — and thus worth the effort necessary to develop parallel, external-memory, lightweight construction algorithms and equally compressible auxiliary data structures. This talk will review the constraints we must adapt to, recent theoretical and practical successes, and some of the targets we should aim for.

summary

Theorem (FM: FOCS '00, JACM 2005)

Given a text $T[1..n]$ and $k \leq (1-\epsilon)\log_\sigma n$, we can store T in $nH_k(T) + o(n \log \sigma)$ bits, where $H_k(T) \leq \lg \sigma$ is the kth-order empirical entropy of T, such that later, given a pattern $P[1..m]$, we can count the occurrences of P in T in $O(m \log\log \sigma)$ time and then report their locations in $O(\log^{1+\epsilon} n)$ time per occurrence.

Bowtie

Bowtie (sequence analysis)

From Wikipedia, the free encyclopedia

Bowtie is a software package commonly used for sequence alignment and sequence analysis in bioinformatics.[1].The source code for the package is distributed freely and compiled binaries are available for Linux, macOS and Windows platforms. As of 2017, the *Genome Biology* paper describing the original Bowtie method has been cited more than 11,000 times.[1] Bowtie is open-source software and is currently maintained by Johns Hopkins University.

Bowtie

Original author(s)	Ben Langmead,Cole Trapnell, Mihai Pop and Steven Salzberg
Developer(s)	Ben Langmead et al.,
Stable release	2.3.4 / 29 December 2017; 6 months ago
Repository	https://github.com/BenLangmead/bowtie
Operating system	Linux, macOS, Windows
Size	14.7 MB (Source)
Type	Bioinformatics
Website	www.bowtie-bio.sourceforge.net

Contents [hide]

History [edit]

The Bowtie sequence aligner was originally developed by Ben Langmead *et al.* at the University of Maryland in 2009.[1] The aligner is typically used with short reads and a large reference genome, or for whole genome analysis. Bowtie is promoted as "an ultrafast, memory-efficient short aligner for short DNA sequences." The speed increase of Bowtie is partly due to implementing the Burrows–Wheeler transform for aligning, which reduces the memory footprint (typically to around 2.2GB for the human genome);[2] a similar method is used by the BWA[3] and SOAP2[4] alignment methods. [2]

Bowtie conducts a quality-aware, greedy, randomized, depth-first search through the space of possible alignments. Because the search is greedy, the first valid alignment encountered by Bowtie will not necessarily be the 'best' in terms of the number of mismatches or in terms of quality.

Bowtie is used as a sequence aligner by a number of other related bioinformatics algorithms, including TopHat,[5] Cufflinks[6] and the CummeRbund Bioconductor package.[7]

Bowtie 2 [edit]

On 16 October 2011, the developers released a beta fork of the project called **Bowtie 2**.[8] In addition to the Burrows-Wheeler transform, Bowtie 2 also uses an FM-index (similar to a suffix array) to keep its memory footprint small. Due to its implementation, Bowtie 2 is more suited to finding longer, gapped alignments in comparison with the original Bowtie method. There is no upper limit on read length in Bowtie 2 and it allows alignments to overlap ambiguous characters in the reference.

CST

With auxiliary data structures that are small relative to the entropy-compressed FM-index, we can support the following operations efficiently:

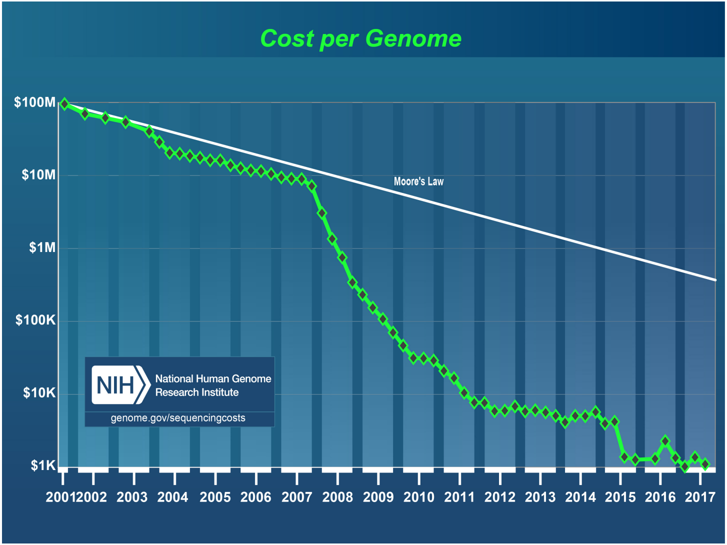
Operation	Description
$\mathit{Root}()$	Suffix tree root.
$\mathit{Locate}(v)$	Text position i of leaf v.
$\mathit{Ancestor}(v, w)$	Whether v is an ancestor of w.
$\mathit{SDepth}(v)$	String depth for internal nodes, i.e., length of string represented by v.
$\mathit{TDepth}(v)$	Tree depth, i.e., depth of tree node v.
$\mathit{Count}(v)$	Number of leaves in the subtree of v.
$\mathit{Parent}(v)$	Parent of v.
$\mathit{FChild}(v)$	First child of v.
$\mathit{NSibling}(v)$	Next sibling of v.
$\mathit{SLink}(v)$	Suffix-link, i.e., if v represents $a \cdot \alpha$ then the node that represents α, for $a \in [1..\sigma]$.
$\mathit{WLink}(v, a)$	Weiner-link, i.e., if v represents α then the node that represents $a \cdot \alpha$.
$\mathit{SLink}^i(v)$	Iterated suffix-link.
$\mathit{LCA}(v, w)$	Lowest common ancestor of v and w.
$\mathit{Child}(v, a)$	Child of v by letter a.
$\mathit{Letter}(v, i)$	The *ith* letter of the string represented by v.
$\mathit{LAQ}_S(v, d)$	String level ancestor, i.e., the highest ancestor of v with string-depth $\geq d$.
$\mathit{LAQ}_T(v, d)$	Tree level ancestor, i.e., the ancestor of v with tree-depth d.

CST

With auxiliary data structures that are small relative to the entropy-compressed FM-index, we can support the following operations efficiently:

Operation	Description
Root()	Suffix tree root.
Locate(v)	Text position i of leaf v.
Ancestor(v, w)	Whether v is an ancestor of w.
SDepth(v)	String depth for internal nodes, i.e., length of string represented by v.
TDepth(v)	Tree depth, i.e., depth of tree node v.
Count(v)	Number of leaves in the subtree of v.
Parent(v)	Parent of v.
FChild(v)	First child of v.
NSibling(v)	Next sibling of v.
SLink(v)	Suffix-link, i.e., if v represents $a \cdot \alpha$ then the node that represents α, for $a \in [1..\sigma]$.
WLink(v, a)	Weiner-link, i.e., if v represents α then the node that represents $a \cdot \alpha$.
$SLink^i(v)$	Iterated suffix-link.
LCA(v, w)	Lowest common ancestor of v and w.
Child(v, a)	Child of v by letter a.
Letter(v, i)	The *ith* letter of the string represented by v.
$LAQ_S(v, d)$	String level ancestor, i.e., the highest ancestor of v with string-depth $\geq d$.
$LAQ_T(v, d)$	Tree level ancestor, i.e., the ancestor of v with tree-depth d.

costs


Cost per Genome
$100M
$10M
$1M
$100K
$10K
$1K
Moore's Law
NIH National Human Genome Research Institute
genome.gov/sequencingcosts
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017

RLFM-index

Theorem (MNSV: SPIRE '08, RECOMB '09, JCB 2010)

Given a text $T[1..n]$ *and a sample rate* d*, we can store* T *in* $O(r + n/d)$ *space, where* r *is the number of runs in the BWT of* T*, such that later, given a pattern* $P[1..m]$*, we can count the occurrences of* P *in* T *in* $O(m \log \log n)$ *time and then report their locations in* $O(d \log \log n)$ *time per occurrence.*

abstract

The Burrows-Wheeler Transform (BWT) is the basis of several tools that have enabled the genomics revolution, but researchers developing those tools are now victims of their own success: next-generation sequencing (NGS) has resulted in genomic databases so large that we strain to build their BWTs, and we can no longer afford the auxiliary data structures we used to take for granted. We are now losing functionality in practice because some of our techniques do not scale. The BWTs themselves remain small and fast and beautiful, however — and thus worth the effort necessary to develop parallel, external-memory, lightweight construction algorithms and equally compressible auxiliary data structures. This talk will review the constraints we must adapt to, recent theoretical and practical successes, and some of the targets we should aim for.

r-index

Theorem (GNP: SODA '18)

We can store a given text $T[1..n]$ *in* $O(r)$ *words, where r is the number of runs in the BWT of* T*, such that later, given a pattern* $P[1..m]$*, we can count the occurrences of* P *in* T *in* $O(m \log \log n)$ *time and then report their locations in* $O(\log \log n)$ *time per occurrence.*

r-index

Theorem (GNP: SODA '18)

We can store a given text $T[1..n]$ in $O(r)$ words, where r is the number of runs in the BWT of T, such that later, given a pattern $P[1..m]$, we can count the occurrences of P in T in $O(m \log\log n)$ time and then report their locations in $O(\log\log n)$ time per occurrence.

Theorem (BGI: CPM '18)

We can prepend a character to T and update the r-index in $O(\log r)$ time.

CST

With auxiliary data structures that fit in $O(r \log(n/r))$ words, we can still support the following operations efficiently:

Operation	Description
$Root()$	Suffix tree root.
$Locate(v)$	Text position i of leaf v.
$Ancestor(v, w)$	Whether v is an ancestor of w.
$SDepth(v)$	String depth for internal nodes, i.e., length of string represented by v.
$TDepth(v)$	Tree depth, i.e., depth of tree node v.
$Count(v)$	Number of leaves in the subtree of v.
$Parent(v)$	Parent of v.
$FChild(v)$	First child of v.
$NSibling(v)$	Next sibling of v.
$SLink(v)$	Suffix-link, i.e., if v represents $a \cdot \alpha$ then the node that represents α, for $a \in [1..\sigma]$.
$WLink(v, a)$	Weiner-link, i.e., if v represents α then the node that represents $a \cdot \alpha$.
$SLink^i(v)$	Iterated suffix-link.
$LCA(v, w)$	Lowest common ancestor of v and w.
$Child(v, a)$	Child of v by letter a.
$Letter(v, i)$	The *ith* letter of the string represented by v.
$LAQ_S(v, d)$	String level ancestor, i.e., the highest ancestor of v with string-depth $\geq d$.
$LAQ_T(v, d)$	Tree level ancestor, i.e., the ancestor of v with tree-depth d.

Langmead et al.