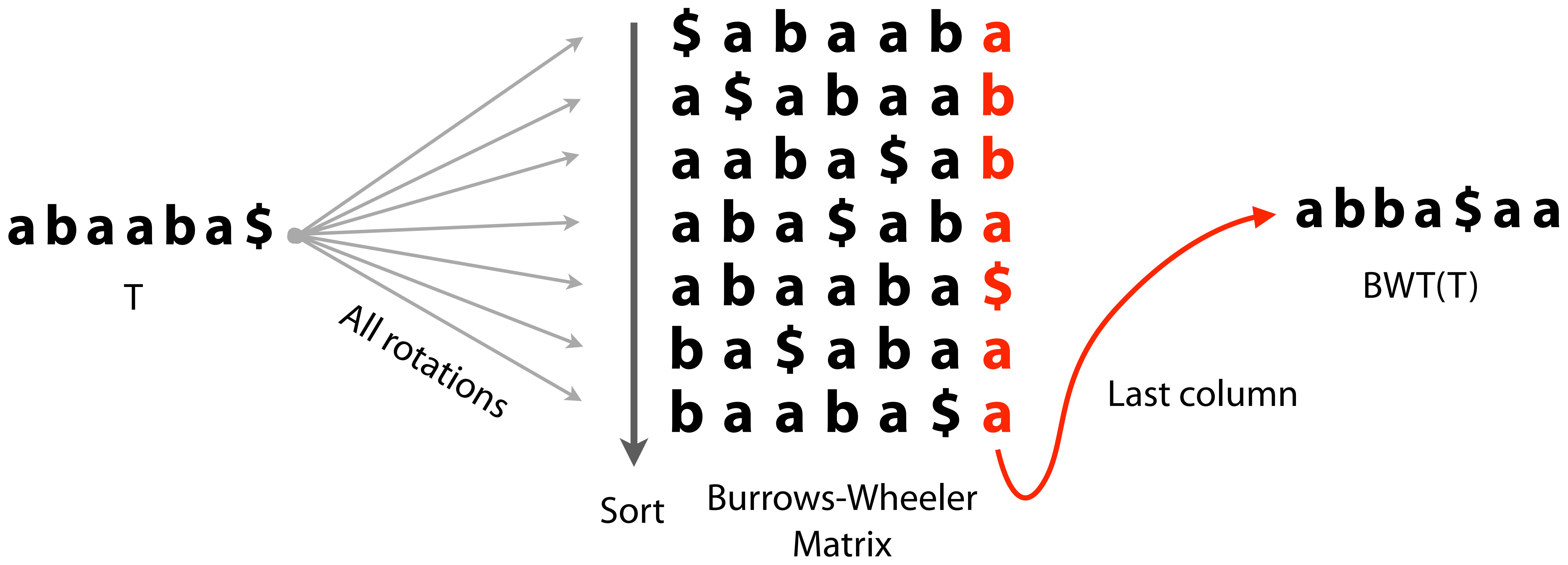
BWT & FM INDEX

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)) ]
```

Make list of all rotations

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

Sort them

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

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_'
```

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	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 $
BWM(T)	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"

$	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

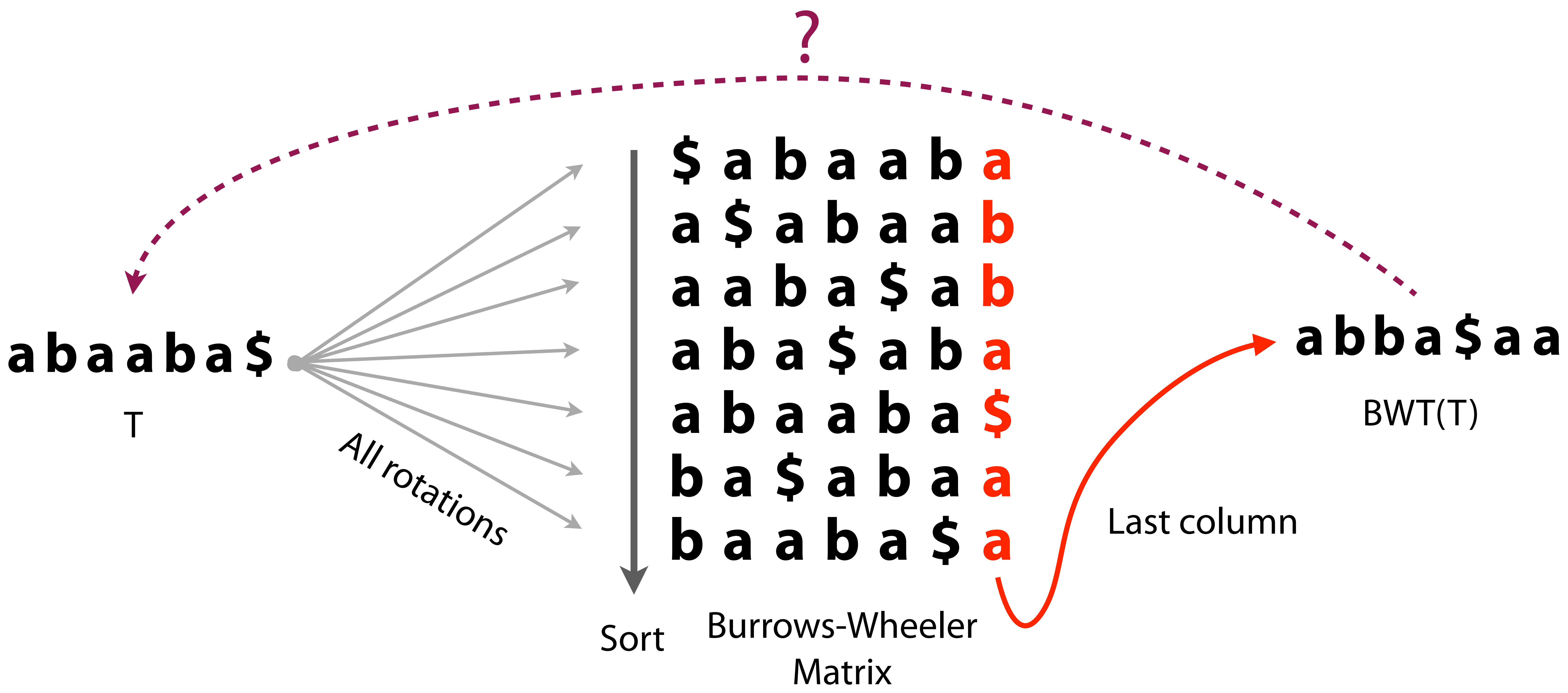
BWM(T)

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)

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...

Note: we *do not* actually write this information in the text / BWM, we Are simply including it here to help us track "which" occurrences of each character in the BWM correspond to the occurrences in the text.

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	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

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

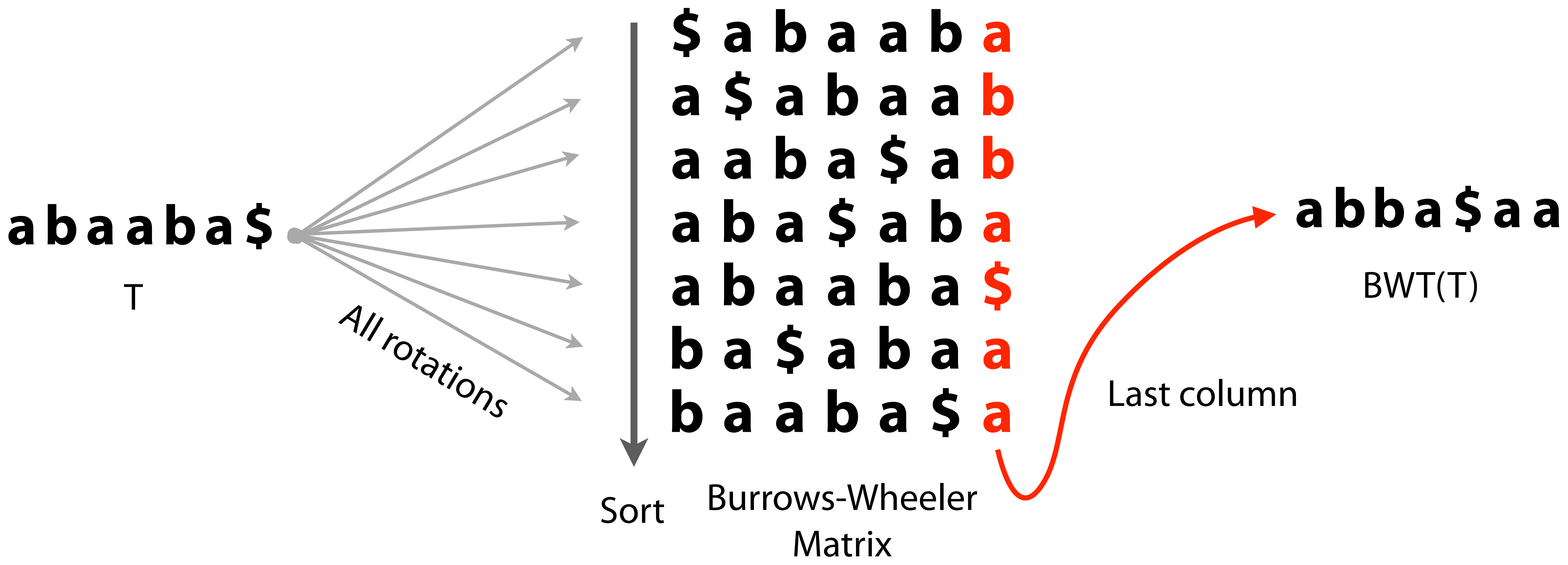
BWM with T-ranking:

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?

\$	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

They're sorted by right-context

\$	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

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:

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

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*	
	$	**a**$_0$	
	a$_0$	**b**$_0$	
	a$_1$	**b**$_1$ ⟵	Which BWM row *begins* with **b**$_1$?
	a$_2$	**a**$_1$	Skip row starting with **$** (1 row)
	a$_3$	$	Skip rows starting with **a** (4 rows)
	b$_0$	**a**$_2$	Skip row starting with **b**$_0$ (1 row)
row 6 ⟶	**b**$_1$	**a**$_3$	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 $\mathbf{G}_{100}$? (Ranks are B-ranks.)

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

Skip rows starting with **A** (300 rows)

Skip rows starting with **C** (400 rows)

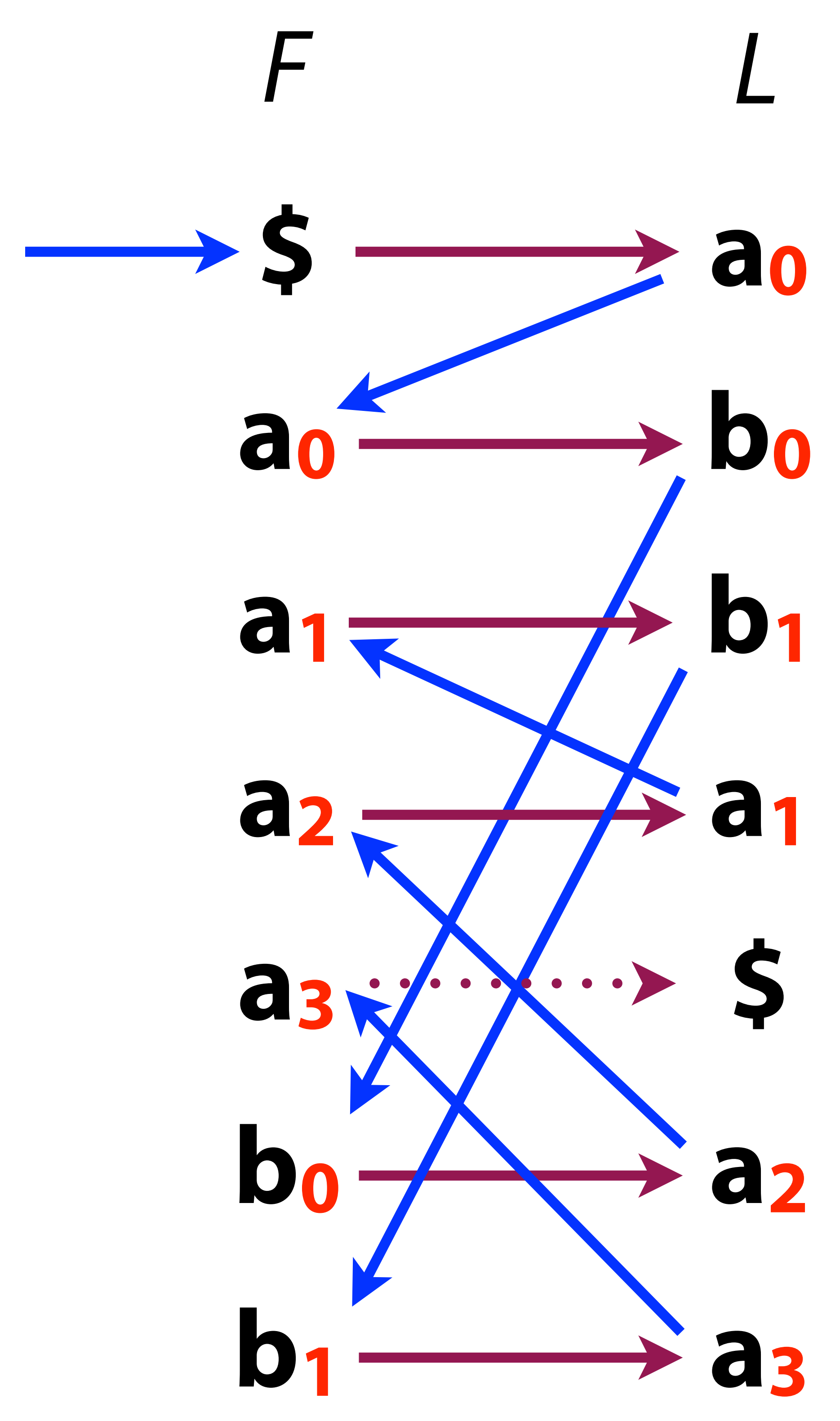
Skip first 100 rows starting with **G** (100 rows)

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 $\mathbf{b}_1$

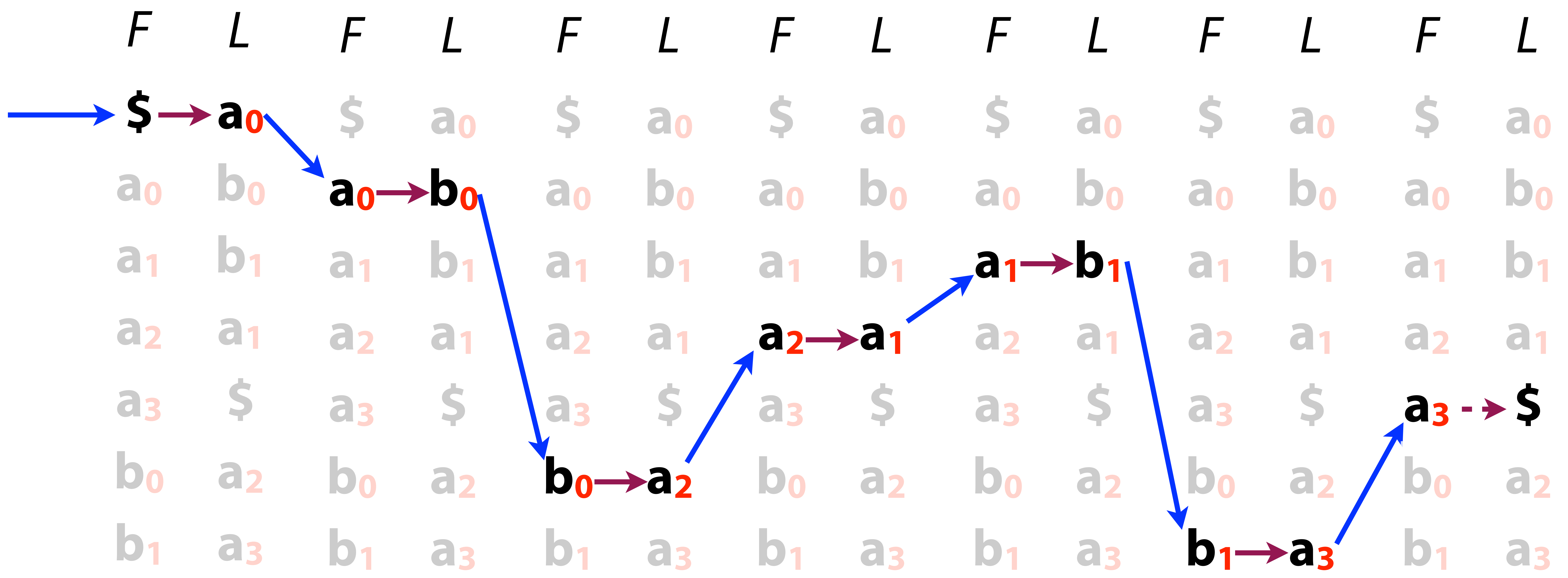
Repeat for $\mathbf{b}_1$, get $\mathbf{a}_3$

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

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

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

```
>>> 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

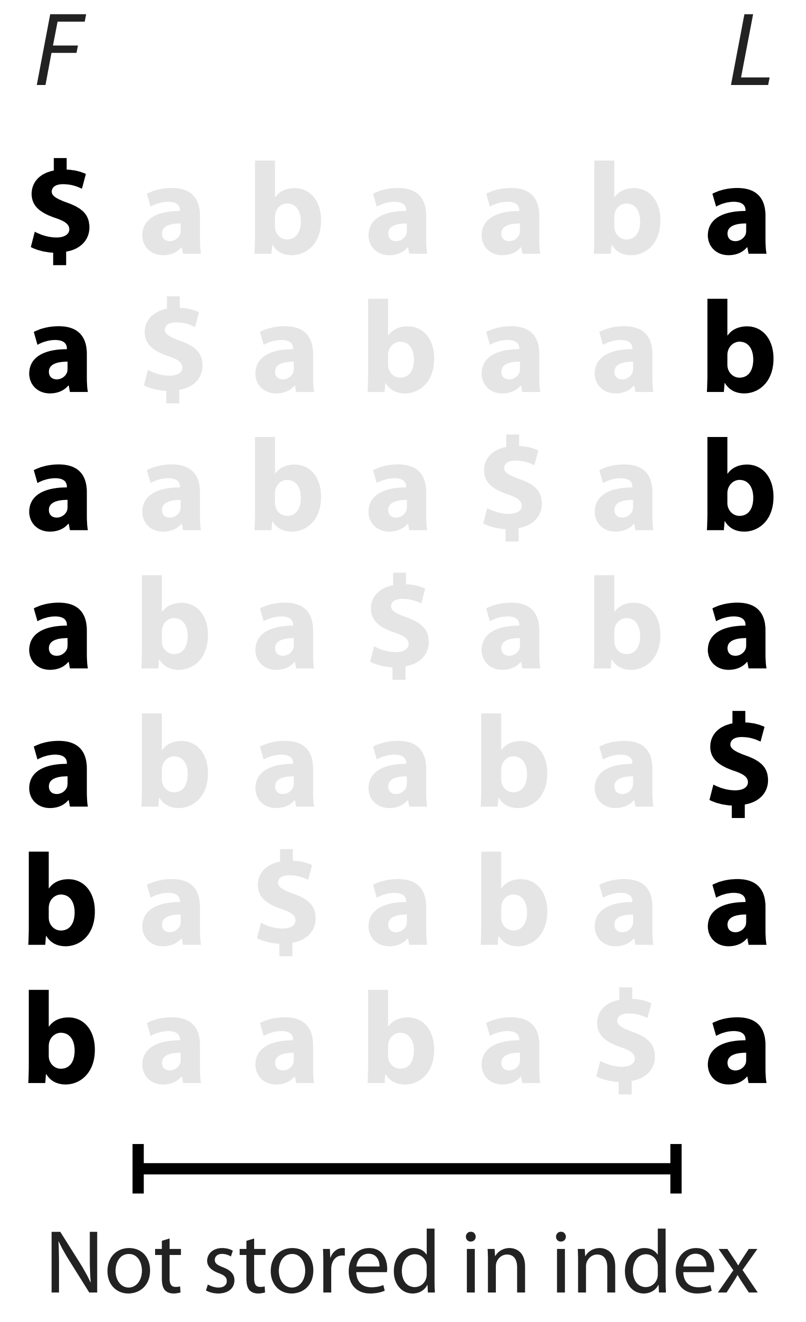
How is it used as an index?

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 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	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

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*.

$\mathbf{b}_0$, $\mathbf{b}_1$ 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**

$P = \mathbf{aba}$

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.

$P = \mathbf{aba}$

Use LF Mapping →

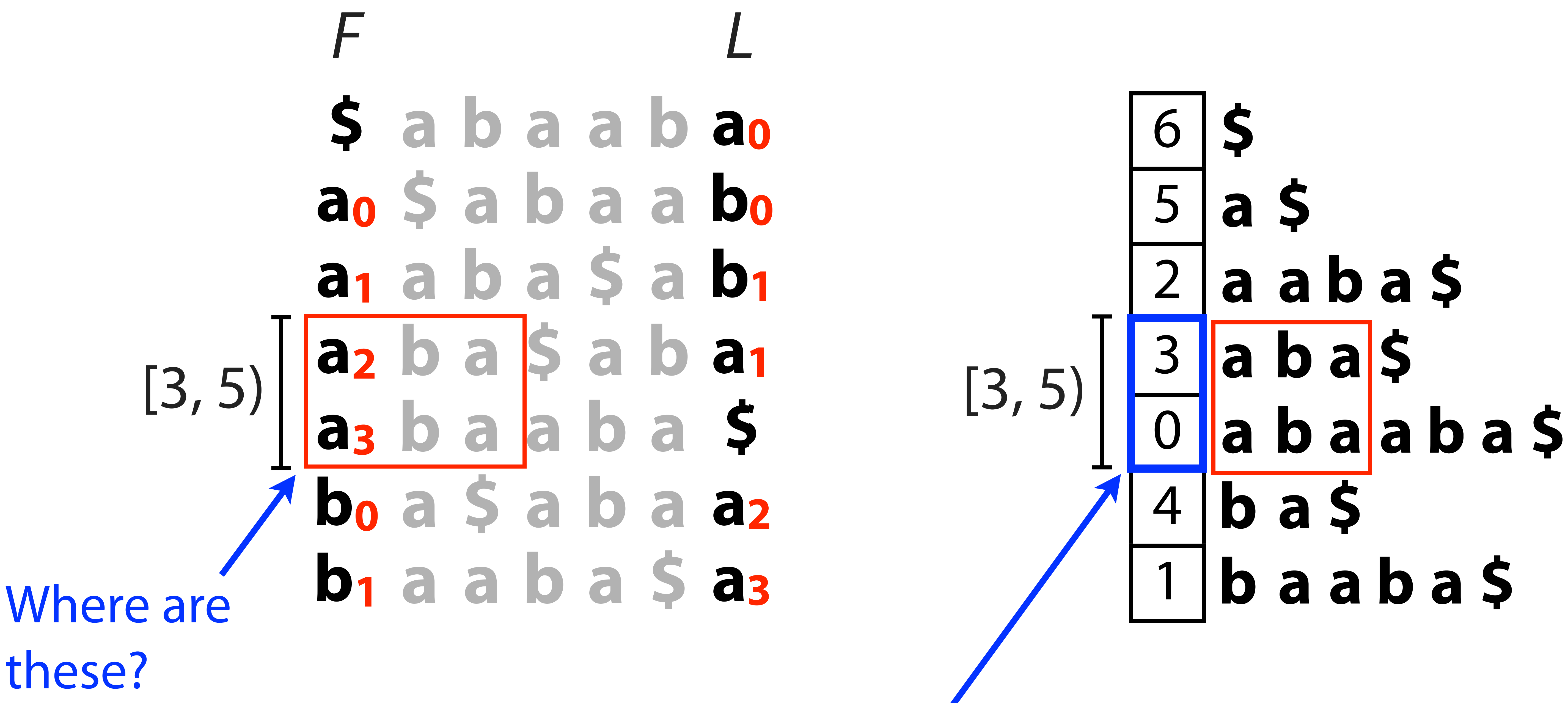
F						L
$\$$	a	b	a	a	b	a_0
a_0	$\$$	a	b	a	a	b_0
a_1	a	b	a	$\$$	a	b_1
a_2	b	a	$\$$	a	b	a_1
a_3	b	a	a	b	a	$\$$
b_0	a	$\$$	a	b	a	a_2
b_1	a	a	b	a	$\$$	a_3

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	$\mathbf{a}_0$
$\mathbf{a}_0$	$	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	$	$\mathbf{a}_3$

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	$\mathbf{a}_3$
$\mathbf{a}_0$	$	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	$	$\mathbf{a}_0$

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	$\mathbf{a}_0$
$\mathbf{a}_0$	$	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	$	$\mathbf{a}_3$

Occ(*c*, *k*) = # of times ***c*** occurs in the first ***k*** characters of BWT(S), aka the LF mapping.

Tally — also referred to as **Occ**(c, k)

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

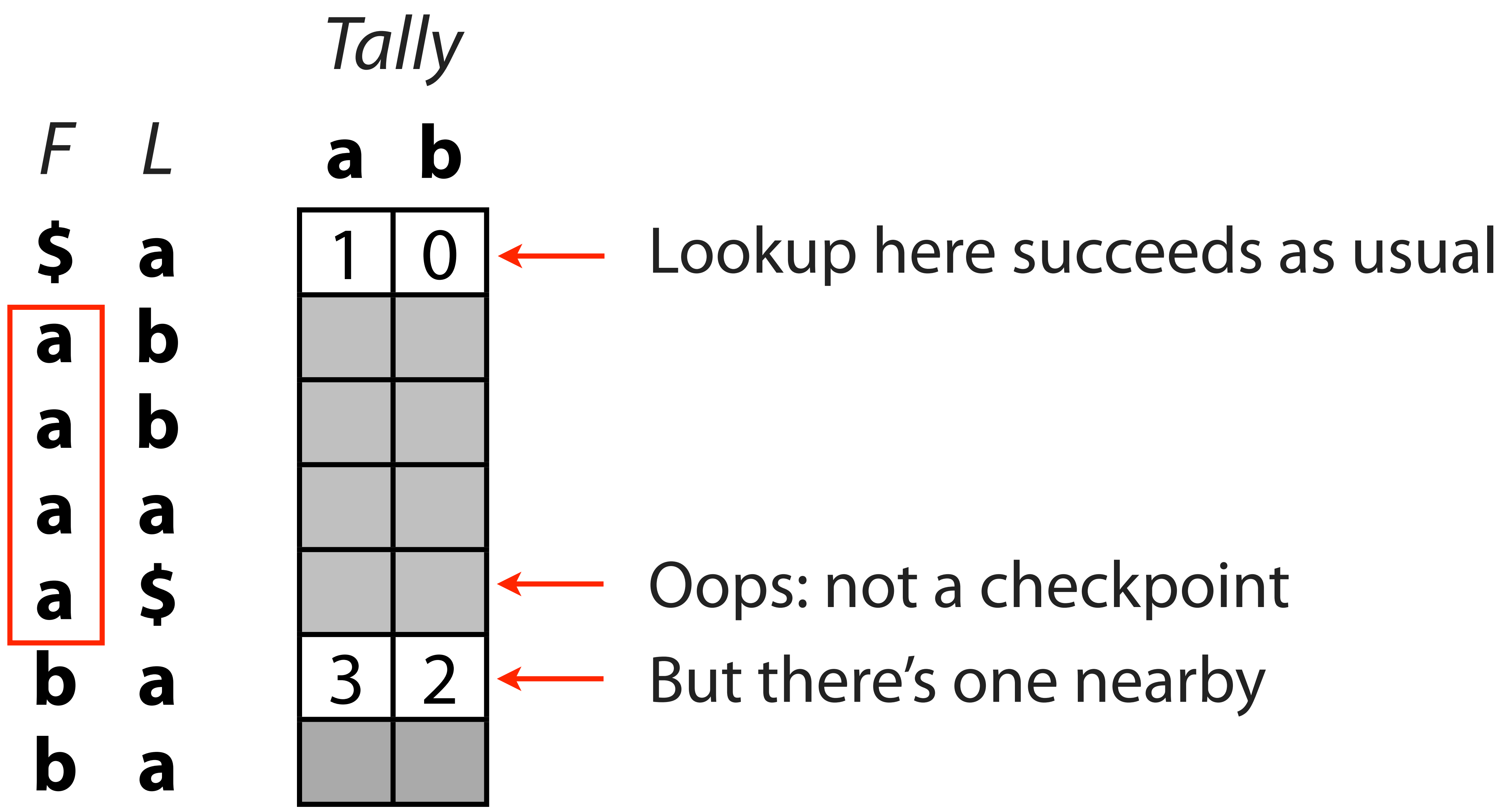
F	*L*	**a**	**b**
$	**a**	1	0
a	**b**	1	1
a	**b**	1	2
a	**a**	2	2
a	**$**	2	2
b	**a**	3	2
b	**a**	4	2

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*.



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 c*s *we saw along the way*.

FM Index: fast rank calculations

What's my rank?

482 + 2 - 1 = 483

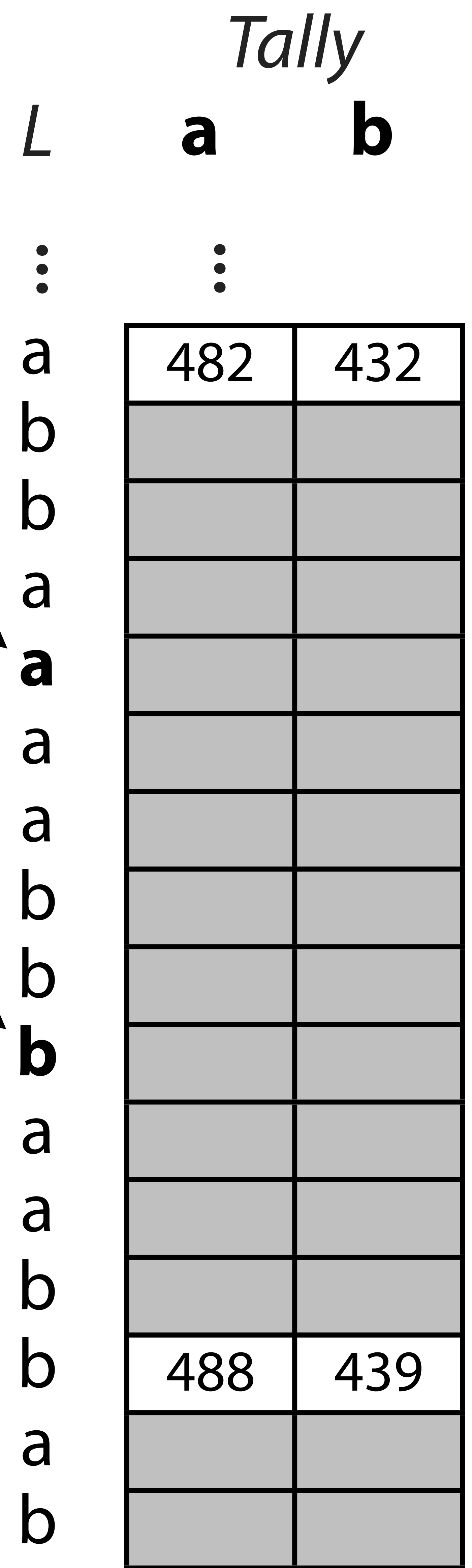
Checkpoint

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: fast rank calculations

This can also be accomplished using **bit-vector rank** operations. We store one bit-vector for each character of Σ, placing a 1 where this character occurs and a 0 everywhere else:

		Tally	
F	*L*	**a**	**b**
\$	**a**	1	0
a	**b**	0	1
a	**b**	0	1
a	**a**	1	0
a	**\$**	0	0
b	**a**	1	0
b	**a**	1	0

the operation **rank(x, i)** returns the total number of 1's in a bit-vector up to (and including) index i. **rank(x,i)** is a *constant-time operation*

rank(a,3) = 2

rank(a,5) = 3

rank(b,5) = 2

To resolve the rank for a given character **c** at a given index **i**, we simply issue a **rank(c,i)** query. This is a practically-fast constant-time operation, but we need to keep around Σ bit-vectors, each of o(m) bits.

FM Index: a few problems

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

(1)

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

This scan is $O(m)$ work

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**

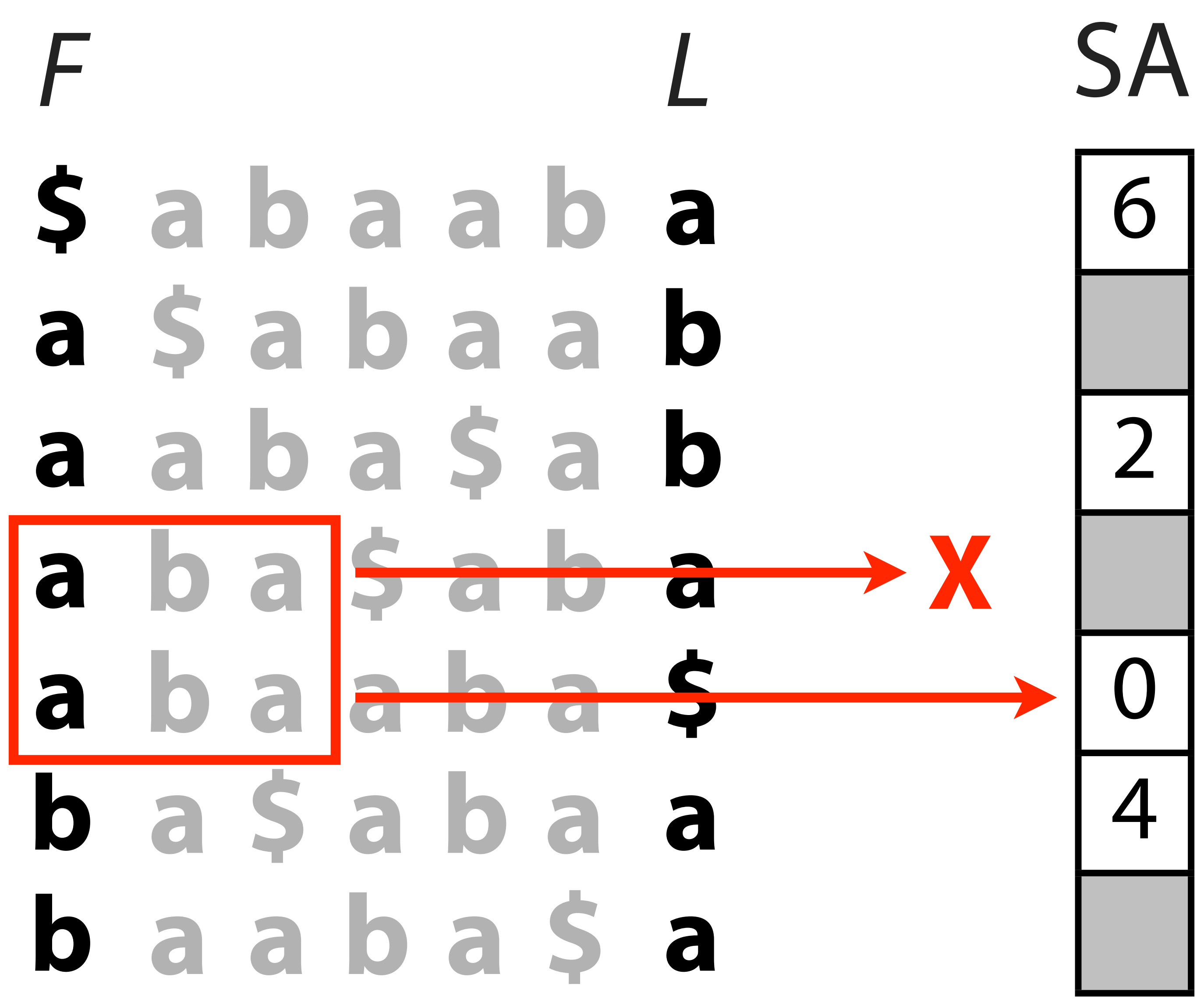
Offsets: 0, 3

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



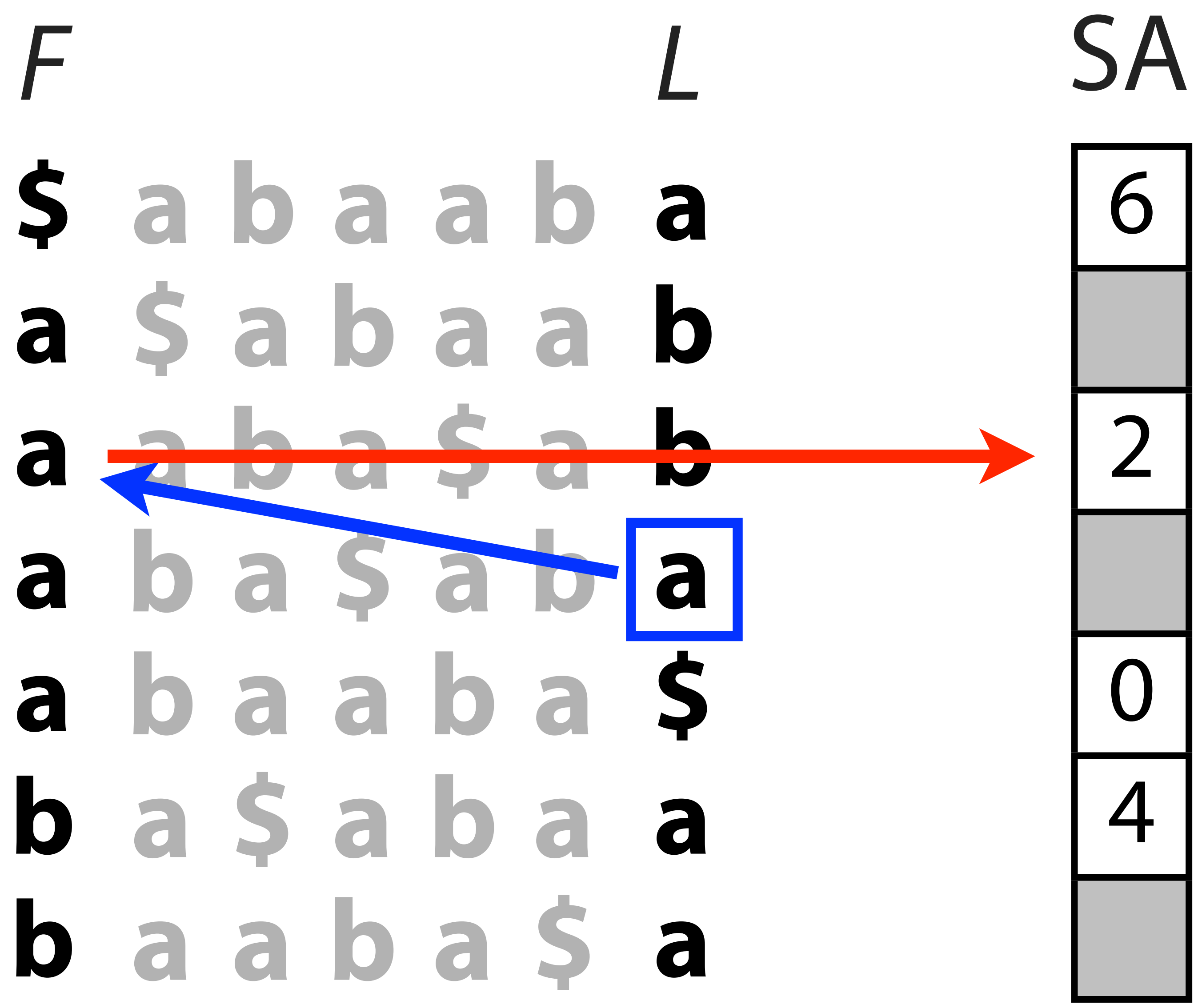
Lookup for row 4 succeeds - we kept that entry of SA

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): $\sim |\Sigma|$ 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 * 3 billion chars = 750 MB

SA sample: 3 billion chars * 4 bytes/char / 32 = ~ 400 MB

Checkpoints: 3 billion * 4 bytes/char * 4 char / 128 = ~400MB

Total ~ 1.5 GB

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.

[1] Kärkkäinen, Juha. "Fast BWT in small space by blockwise suffix sorting." *Theoretical Computer Science* 387.3 (2007): 249-257.

*slide courtesy of Carl Kingsford