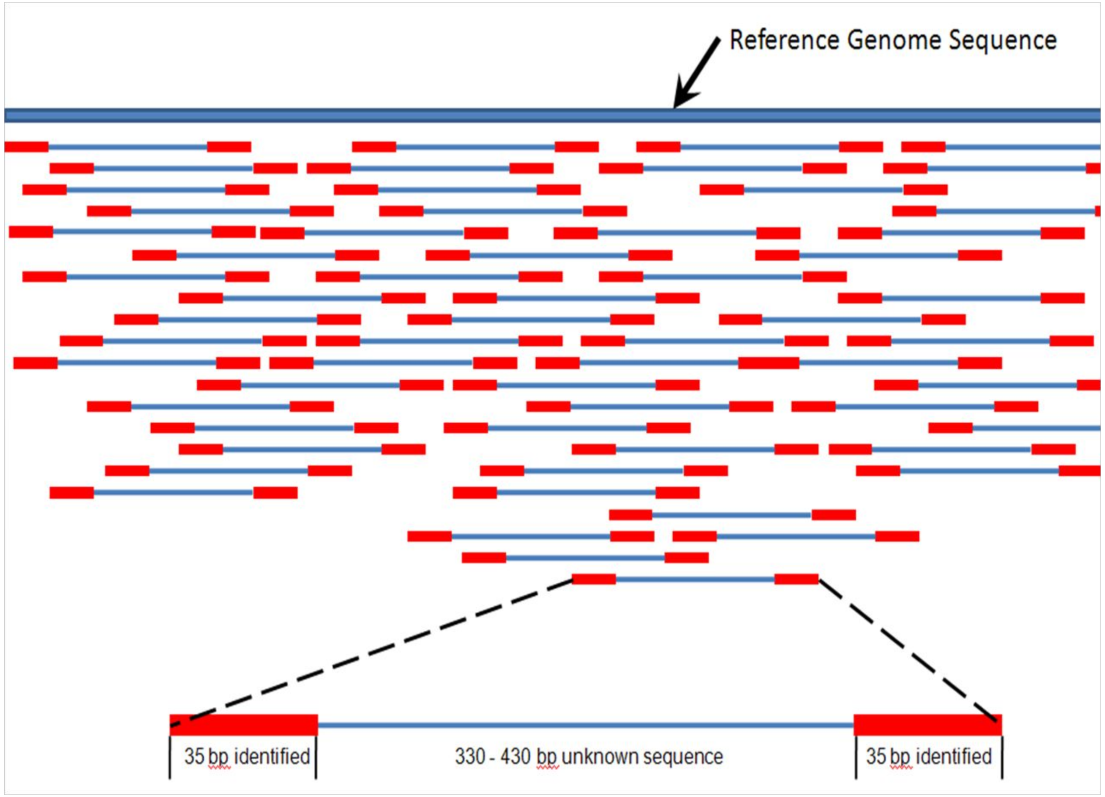
Burrows-Wheeler Transform and FM Index

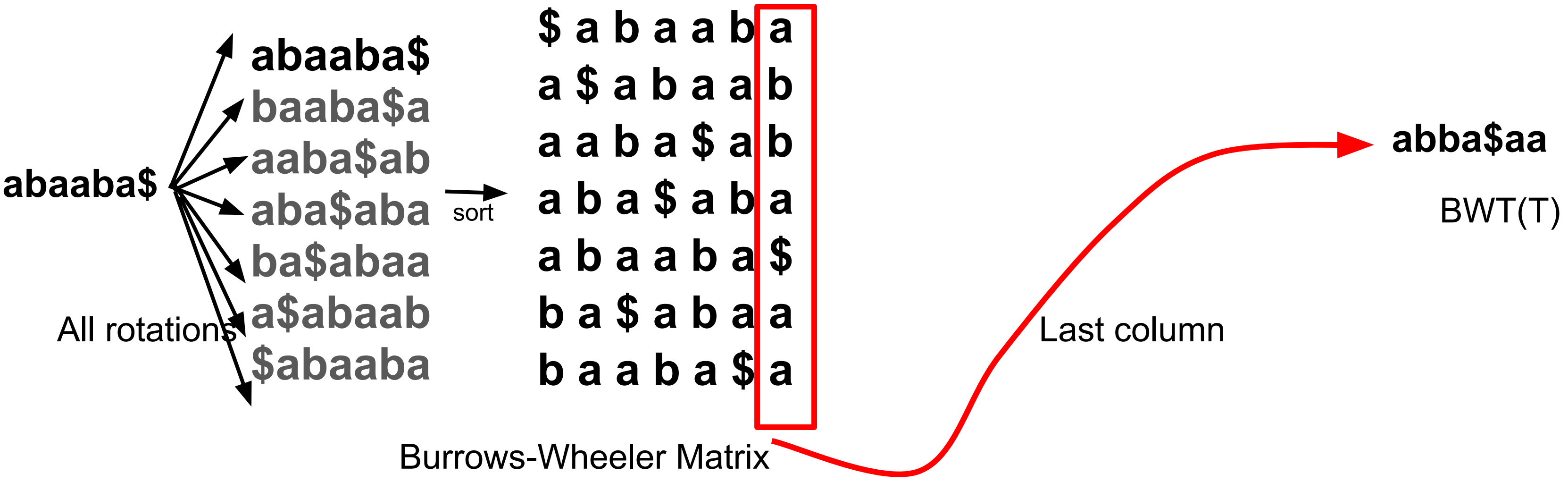
Lesson 08

Recapitulation



Burrows-Wheeler Transform

Burrows-Wheeler Transform



abaaba$

All rotations

abaaba$
baaba$a
aaba$ab
aba$aba
ba$abaa
a$abaab
$abaaba

sort

$abaaba
a$abaab
aaba$ab
aba$aba
abaaba$
ba$abaa
baaba$a

Burrows-Wheeler Matrix

Last column

abba$aa

BWT(T)

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

Slide adapted from Ben Langmead

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 range(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 creating BWM """
    return ''.join(map(lambda x: x[-1], bwm(t)))
```

Make list of all rotations

Sort them

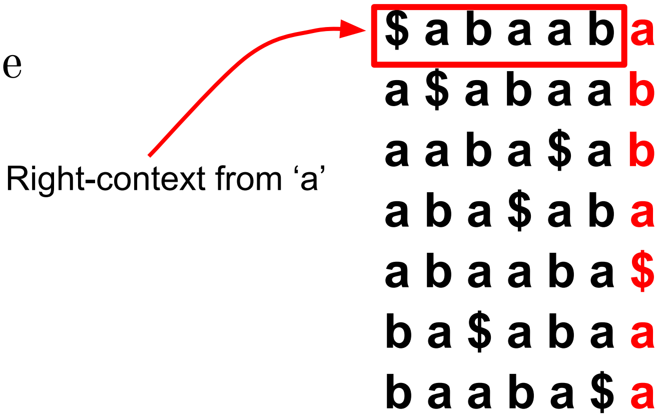
Take last column

```
>>>  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_'
```

Slide adapted from Ben Langmead

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



Right-context from ‘a’

```
$ 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
```

Burrows-Wheeler Matrix

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

Slide adapted from Ben Langmead

Burrows-Wheeler Transform

BWM bears a resemblance to the suffix array

```
$abaaba
a$abaab
aaba$ab
aba$aba
abaaba$
ba$abaa
baaba$a
```

BWT(T)

6	\$
5	a\$
2	aaba\$
3	aba\$
0	abaaba\$
4	ba\$
1	baaba\$

SA(T)

Which structure is very similar to BWM?

Sort order is the same whether rows are rotations or suffixes

Slide adapted from Ben Langmead

Burrows-Wheeler Transform

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

BWT(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 $

BWT = characters just to the left of the suffixes in the suffix array

Slide adapted from Ben Langmead

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 range(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
```

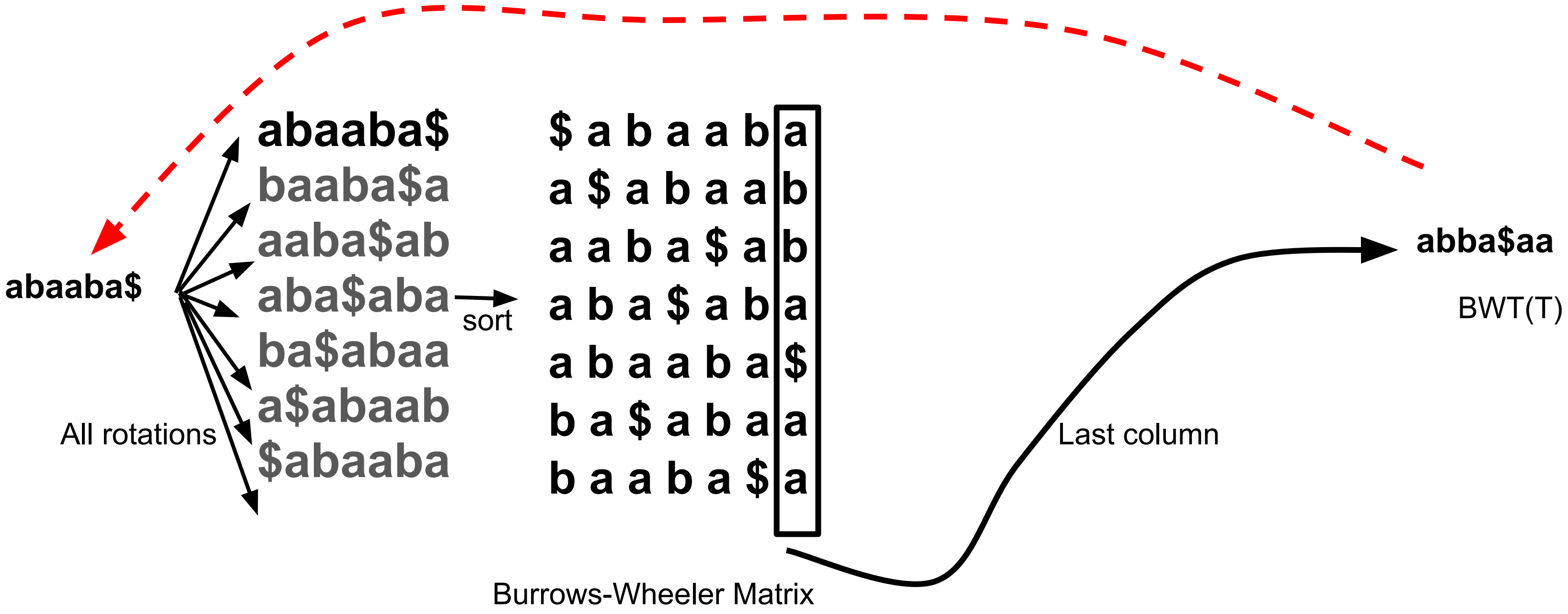
Make suffix array

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_'
```

Slide adapted from Ben Langmead

Burrows-Wheeler Transform



abaaba$
baaba$a
aaba$ab
aba$aba
ba$abaa
a$abaab
$abaaba

```
$ 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
```

How to reverse the BWT? BWM has a key property called the LF Mapping

Slide adapted from Ben Langmead

Burrows-Wheeler Transform: T-ranking

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

$$\mathbf{a}_{0}\ \mathbf{b}_{0}\ \mathbf{a}_{1}\ \mathbf{a}_{2}\ \mathbf{b}_{1}\ \mathbf{a}_{3}\ \mathbf{\$}$$

Now let’s rewrite the BWM including ranks....

Slide adapted from Ben Langmead

Burrows-Wheeler Transform: T-ranking

BWT with T-raking:

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

Look at first and last columns, called F and L

“a” occur in the same order in F and L

As we look down columns, in both cases we see: $\mathbf{a_3, a_1, a_2, a_0}$

Slide adapted from Ben Langmead

Burrows-Wheeler Transform: T-ranking

BWT with T-raking:

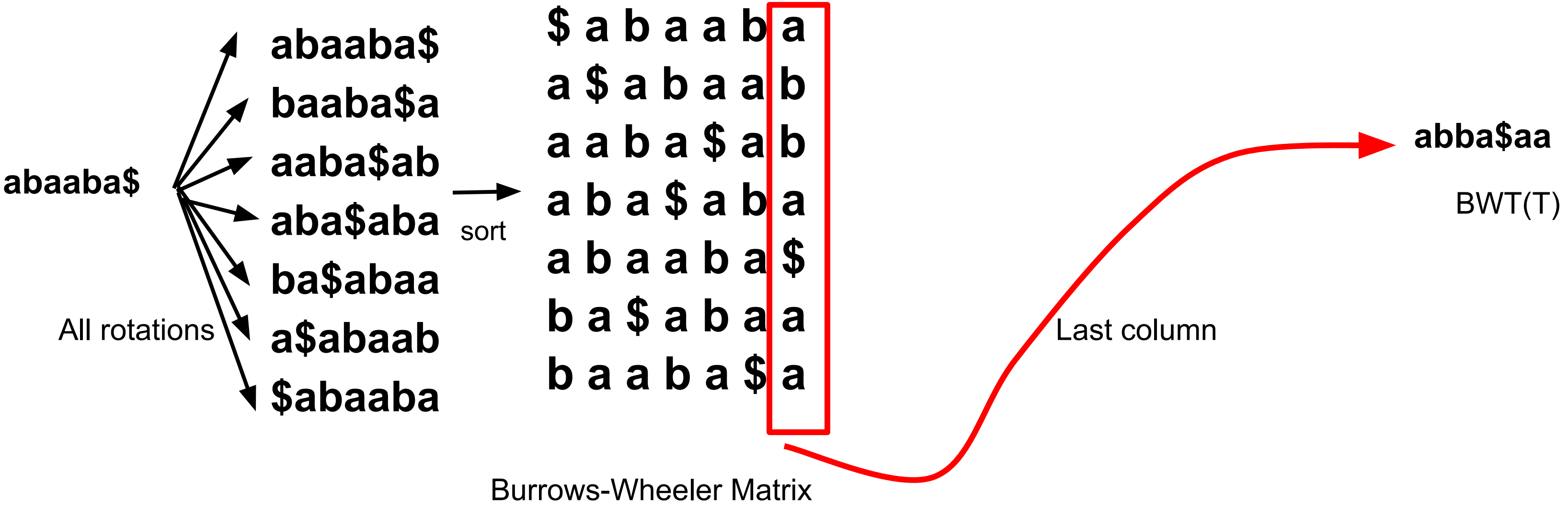
$$
\begin{array}{ccccccc}
\mathbf{F} & & & & & & \mathbf{L} \\
\$ & a_0 & b_0 & a_1 & a_2 & b_1 & a_3 \\
a_3 & \$ & a_0 & b_0 & a_1 & a_2 & \boxed{b_1} \\
a_1 & a_2 & b_1 & a_3 & \$ & a_0 & \boxed{b_0} \\
a_2 & b_1 & a_3 & \$ & a_0 & b_0 & a_1 \\
a_0 & b_0 & a_1 & a_2 & b_1 & a_3 & \$ \\
\boxed{b_1}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

Same is with "b"

Slide adapted from Ben Langmead

Burrows-Wheeler Transform

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



abaaba$

All rotations

abaaba$
baaba$a
aaba$ab
aba$aba
ba$abaa
a$abaab
$abaaba

sort

$ 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

Burrows-Wheeler Matrix

Last column

abba$aa

BWT(T)

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

Slide adapted from Ben Langmead

Burrows-Wheeler Transform: LF Mapping

BWT with T-raking:

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

Order of ranks in L
is preserved in F!

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

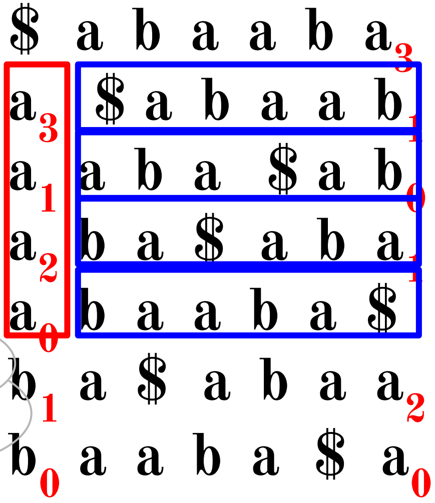
However we rank occurrences of c, ranks appear in the same order in F and L

Slide adapted from Ben Langmead

Burrows-Wheeler Transform: LF Mapping

Why does the LF Mapping hold?
Why are these “a” in this order relative to each other?

They’re sorted by right-context!!!



$\$\ a\ b\ a\ a\ b\ a_3$

$a_3\ \$\ a\ b\ a\ a\ b_1$

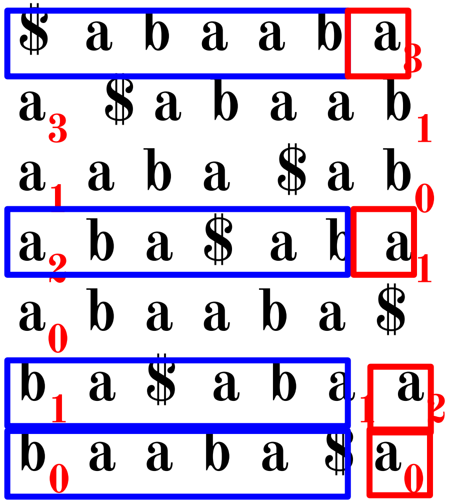
$a_1\ a\ b\ a\ \$\ a\ b_0$

$a_2\ b\ a\ \$\ a\ b\ a_1$

$a_0\ b\ a\ a\ b\ a\ \$$

$b_1\ a\ \$\ a\ b\ a\ a_2$

$b_0\ a\ a\ b\ a\ \$\ a_0$



$\$\ a\ b\ a\ a\ b\ a_3$

$a_3\ \$\ a\ b\ a\ a\ b_1$

$a_1\ a\ b\ a\ \$\ a\ b_0$

$a_2\ b\ a\ \$\ a\ b\ a_1$

$a_0\ b\ a\ a\ b\ a\ \$$

$b_1\ a\ \$\ a\ b\ a\ a_2$

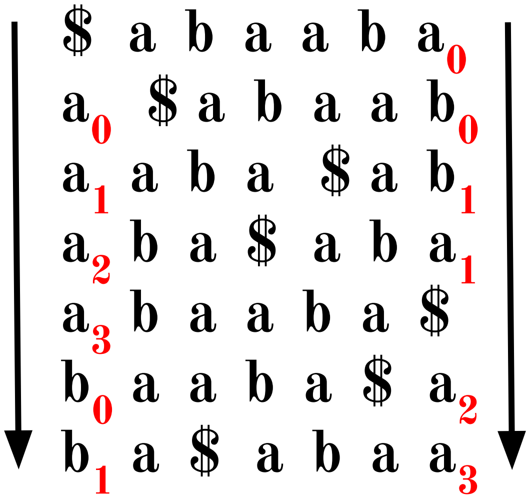
$b_0\ a\ a\ b\ a\ \$\ a_0$

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

Slide adapted from Ben Langmead

Burrows-Wheeler Transform: LF Mapping

BWM with B-ranking:



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

Ascending rank

F now has very simple structure: a \$, a block of “a” with ascending ranks, a block of “b” with ascending ranks (we do not have to store its ranks)

Slide adapted from Ben Langmead

Burrows-Wheeler Transform

	F	L
	$	a_0
	a_0	b_0
	a_1	b_1
	a_2	a_1
	a_3	$
	b_0	a_2
row 6	b_1	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

Slide adapted from Ben Langmead

Burrows-Wheeler Transform

Say T has 300 As, 400 Cs, 250 Gs and 700 Ts and $\$ < A < C < G < T$
Which BWM row (0-based) begins with G_{100}? (Ranks are B-ranks.)

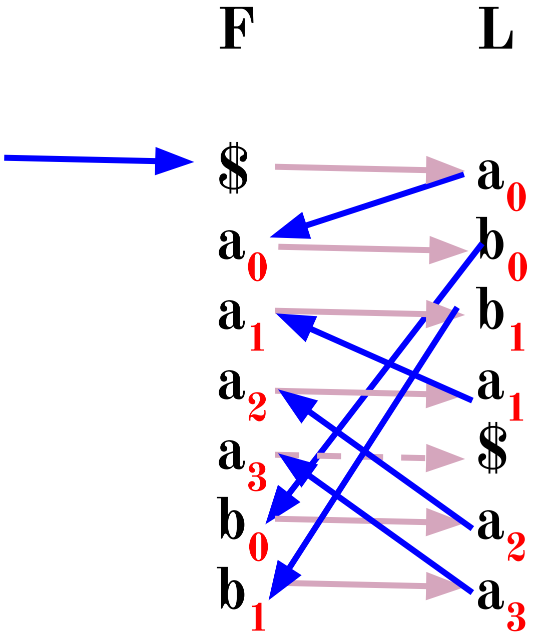
- Skip row starting with \$ (1 row)
- Skip rows starting with A (300 rows)
- Skip rows starting with C (400 rows)
- Skip first 100 rows starting with G (100 rows)
- Answer: row $1 + 300 + 400 + 100 =$ **row 801**

Slide adapted from Ben Langmead

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
...

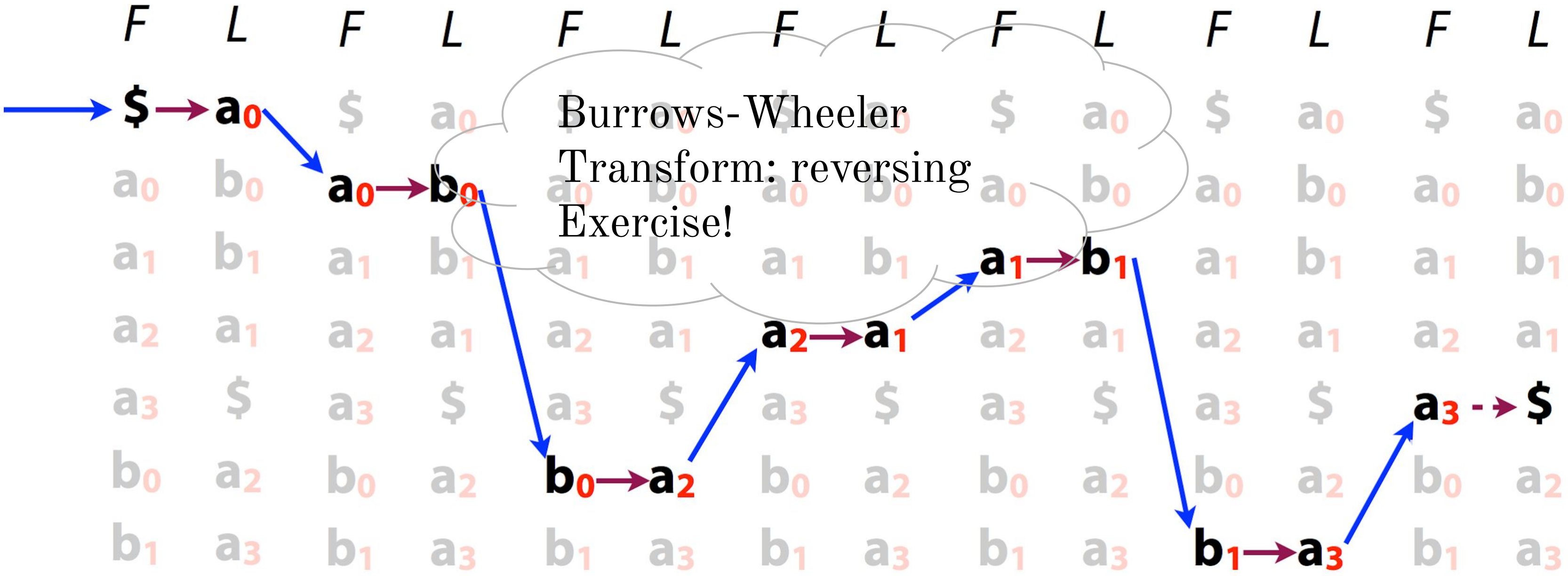


Reverse of chars we visited = $a_3\ b_1\ a_1\ a_2\ b_0\ a_0$ \$ = T

Slide adapted from Ben Langmead

Burrows-Wheeler Transform: reversing

Another way of visualizing Reverse BWT(T)


F L F L F L F L F L F L F L
$ a₀ $ a₀ $ a₀ $ a₀ $ a₀ $ a₀ $ a₀
a₀ b₀ a₀ b₀ a₀ b₀ a₀ b₀ a₀ b₀ a₀ b₀ a₀ b₀
a₁ b₁ a₁ b₁ a₁ b₁ a₁ b₁ a₁ b₁ a₁ b₁ a₁ b₁
a₂ a₁ a₂ a₁ a₂ a₁ a₂ a₁ a₂ a₁ a₂ a₁ a₂ a₁
a₃ $ a₃ $ a₃ $ a₃ $ a₃ $ a₃ $ a₃ $
b₀ a₂ b₀ a₂ b₀ a₂ b₀ a₂ b₀ a₂ b₀ a₂ b₀ a₂
b₁ a₃ b₁ a₃ b₁ a₃ b₁ a₃ b₁ a₃ b₁ a₃ b₁ a₃

Burrows-Wheeler Transform: reversing
Exercise!

Slide adapted from Ben Langmead

Burrows-Wheeler Transform: reversing

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? How to query?

Slide adapted from Ben Langmead

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)

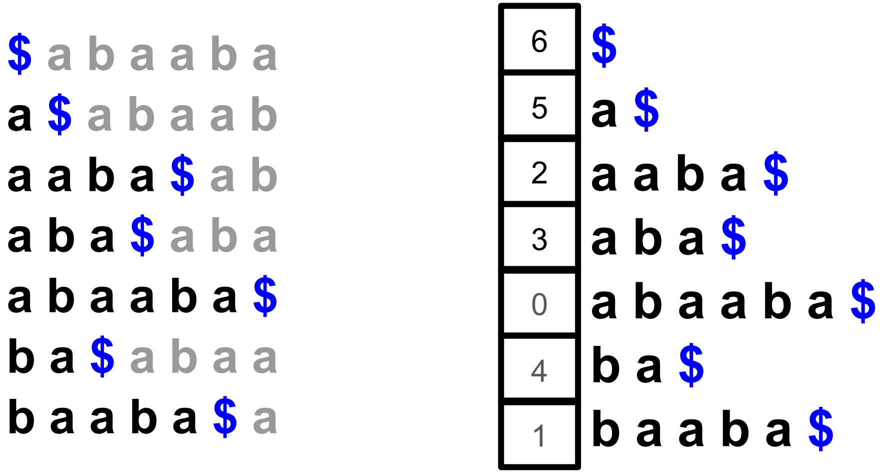
- Core of index consists of F and L from BWM:
    - F can be represented very simply (1 integer per alphabet character)
    - And L is compressible
    - Potentially very space-economical!

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

Slide adapted from Ben Langmead

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 $

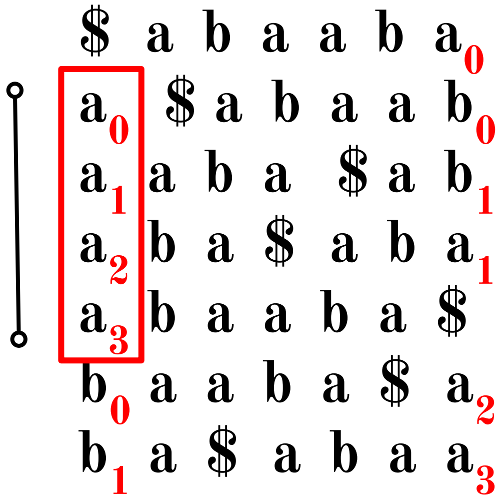
We don’t have these columns; binary search isn’t possible

Slide adapted from Ben Langmead

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

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



$\$\ a\ b\ a\ a\ b\ a_0$

$a_0\ \$\ a\ b\ a\ a\ b_0$

$a_1\ a\ b\ a\ \$\ a\ b_1$

$a_2\ b\ a\ \$\ a\ b\ a_1$

$a_3\ b\ a\ a\ b\ a\ \$$

$b_0\ a\ a\ b\ a\ \$\ a_2$

$b_1\ a\ \$\ a\ b\ a\ a_3$

$\mathbf{P = aba}$

Slide adapted from Ben Langmead

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

$	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 occurring just to left.

P = aba

$	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

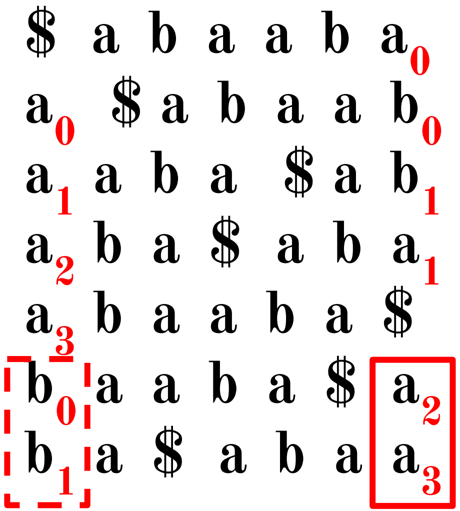
Use LF Mapping. Let new range delimit those b-s

Slide adapted from Ben Langmead

FM Index: querying

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

P = aba



$\$\ a\ b\ a\ a\ b\ a_0$

$a_0\ \$\ a\ b\ a\ a\ b_0$

$a_1\ a\ b\ a\ \$\ a\ b_1$

$a_2\ b\ a\ \$\ a\ b\ a_1$

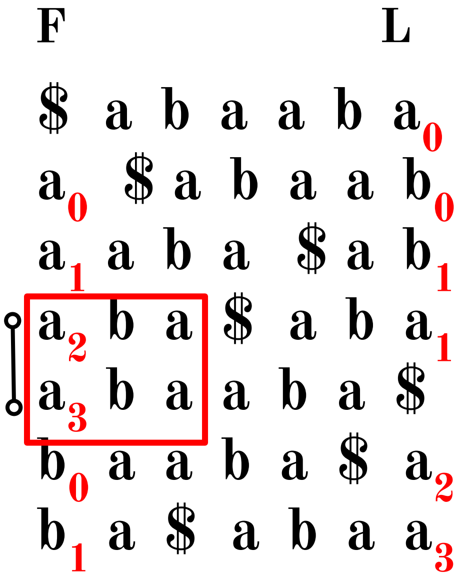
$a_3\ b\ a\ a\ b\ a\ \$$

$b_0\ a\ a\ b\ a\ \$\ a_2$

$b_1\ a\ \$\ a\ b\ a\ a_3$

Occurs just to the 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

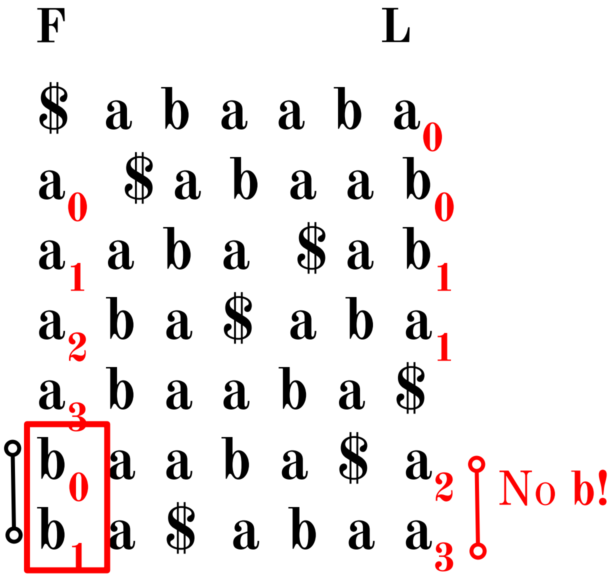
Now we have the rows with prefix **aba**

Slide adapted from Ben Langmead

FM Index: querying

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

$$\mathbf{P = bba}$$



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

Use LF mapping

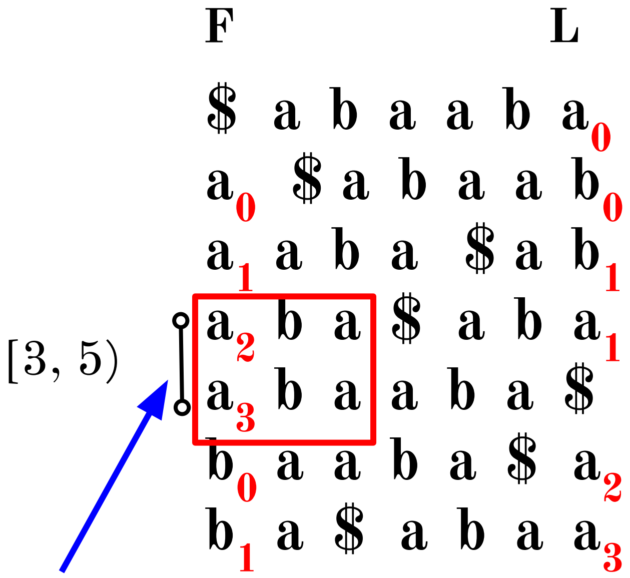
No b!

Slide adapted from Ben Langmead

FM Index: querying

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

$$\mathbf{P} = \mathbf{aba}$$



$\$ \; a \; b \; a \; a \; b \; a_0$

$a_0 \; \$ \; a \; b \; a \; a \; b_0$

$a_1 \; a \; b \; a \; \$ \; a \; b_1$

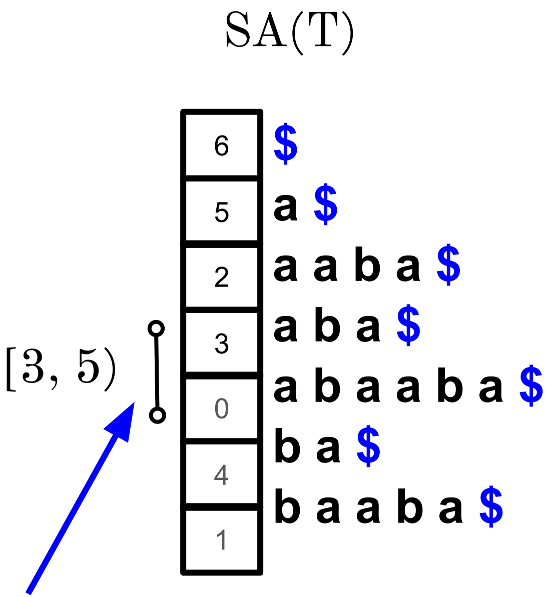
$a_2 \; b \; a \; \$ \; a \; b \; a_1$

$a_3 \; b \; a \; a \; b \; a \; \$$

$b_0 \; a \; a \; b \; a \; \$ \; a_2$

$b_1 \; a \; \$ \; a \; b \; a \; a_3$

Where are these?



SA(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 $

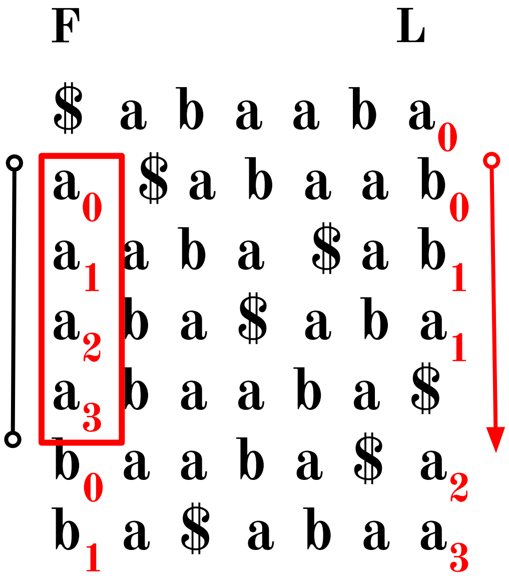
Unlike suffix array, we don't immediately know where the matches are in T...

Slide adapted from Ben Langmead

FM Index: querying

$\mathbf{P = aba}$

Use LF mapping



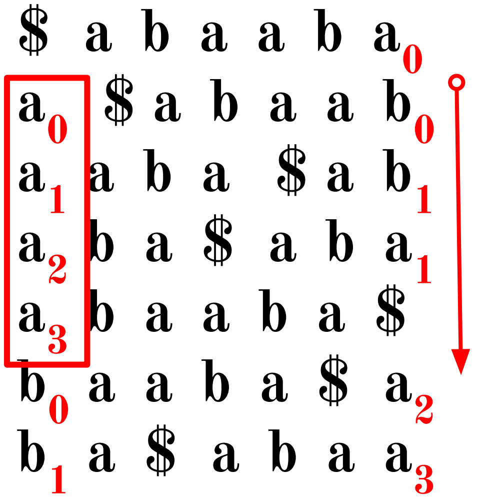
$$\begin{array}{ccccccc}
\mathbf{F} & & & & & & \mathbf{L} \\
\$ & a & b & a & a & b & a_0 \\
\boxed{a_0} & \$ & a & b & a & a & b_0 \\
\boxed{a_1} & a & b & a & \$ & a & b_1 \\
\boxed{a_2} & b & a & \$ & a & b & a_1 \\
\boxed{a_3} & b & a & a & b & a & \$ \\
b_0 & a & a & b & a & \$ & a_2 \\
b_1 & a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

Scan looking for **b** takes too long!

Slide adapted from Ben Langmead

FM Index: Current issues

(1) Scanning for preceding character is slow



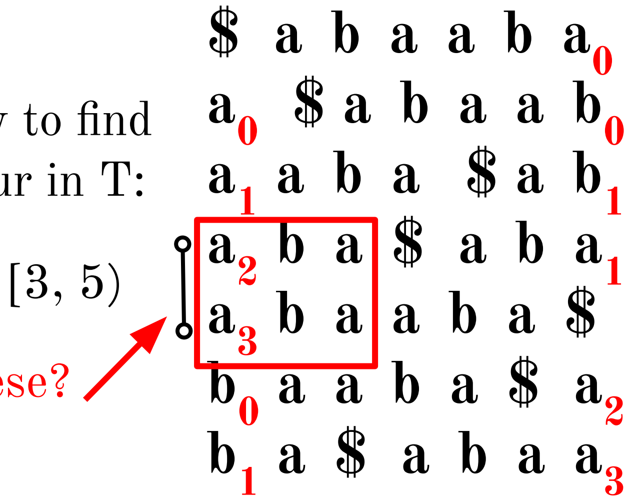
$$\begin{array}{ccccccc}
\$ & a & b & a & a & b & a_0 \\
a_0 & \$ & a & b & a & a & b_0 \\
a_1 & a & b & a & \$ & a & b_1 \\
a_2 & b & a & \$ & a & b & a_1 \\
a_3 & b & a & a & b & a & \$ \\
b_0 & a & a & b & a & \$ & a_2 \\
b_1 & a & \$ & a & b & a & a_3
\end{array}$$

(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 # 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
```

(3) We need a way to find where matches occur in T:



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

[3, 5)

Where are these?

Slide adapted from Ben Langmead

FM Index: fast rank calculations

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

$\$$	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:

F	**L**	**Tally**	
		a	**b**
$\$$	a_0	1	0
a_0	b_0	1	1
a_1	b_1	1	2
a_2	a_1	2	2
a_3	$\$$	2	2
b_0	a_2	3	2
b_1	a_3	4	2

We infer b_0 and b_1 appear in L in this range:

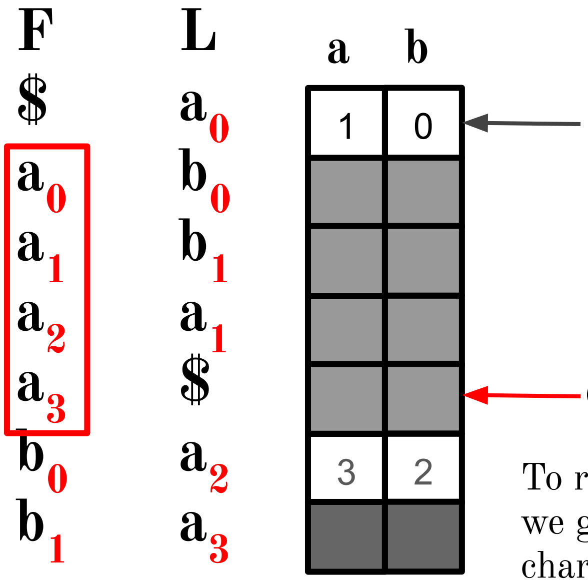
Tally(b, i) - Tally(b,j) = 2

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

Slide adapted from Ben Langmead

FM Index: fast rank calculations

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



F	L	a	b
$	a_0	1	0
a_0	b_0		
a_1	b_1		
a_2	a_1		
a_3	$		
b_0	a_2	3	2
b_1	a_3		

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 characters we saw along the way.

Slide adapted from Ben Langmead

FM Index: fast rank calculations

What’s my rank?

$482 + 2 - 1 = 483$

checkpoint + a-s along the way - tally->rank

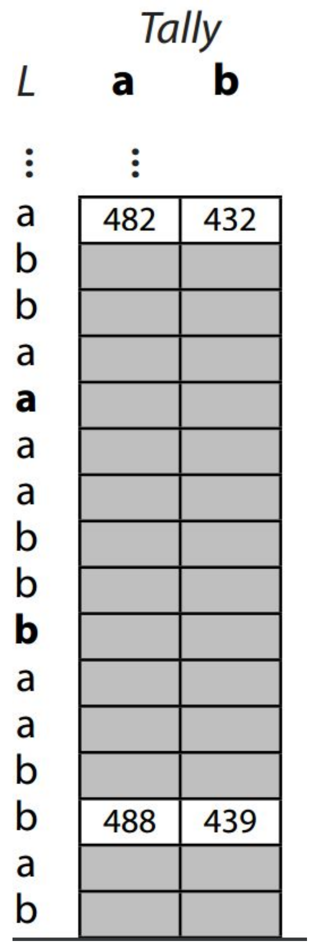
What’s my rank?

$439 - 2 - 1 = 436$

checkpoint + b-s along the way - tally->rank

What’s my rank?

$439 - 2 - 1 = 436$



Tally

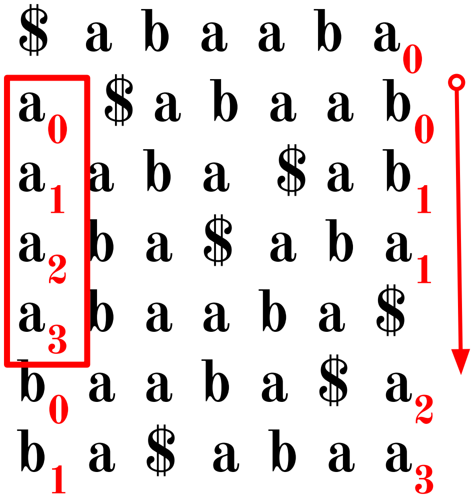
L	**a**	**b**
⋮	⋮	
a	482	432
b		
b		
a		
a		
a		
a		
b		
b		
b		
a		
a		
b		
b	488	439
a		
b		

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

Slide adapted from Ben Langmead

FM Index: Current issues

(1) Scanning for preceding character is slow



$\$\ a\ b\ a\ a\ b\ a_0$

$a_0\ \$\ a\ b\ a\ a\ b_0$

$a_1\ a\ b\ a\ \$\ a\ b_1$

$a_2\ b\ a\ \$\ a\ b\ a_1$

$a_3\ b\ a\ a\ b\ a\ \$$

$b_0\ a\ a\ b\ a\ \$\ a_2$

$b_1\ a\ \$\ a\ b\ a\ a_3$

With checkpoints it’s O(1)

(2) Storing ranks takes too much space

```
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
```

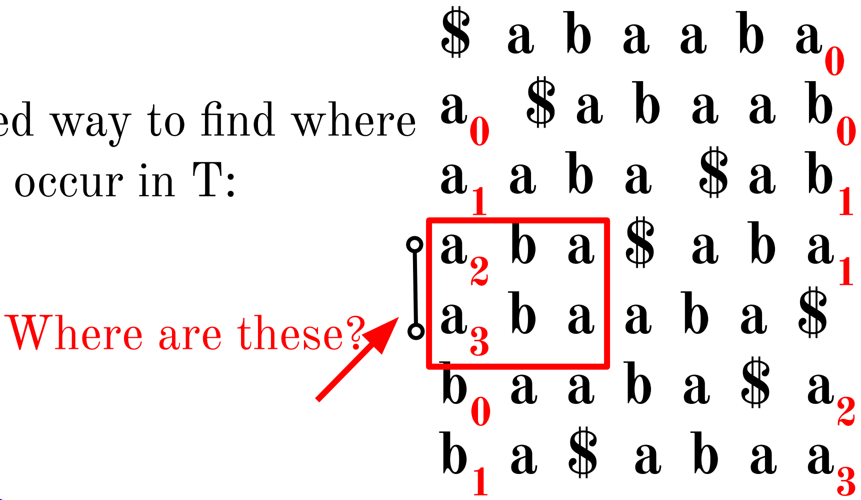
m integers

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

Slide adapted from Ben Langmead

FM Index: Not yet solved!

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



$\$\ a\ b\ a\ a\ b\ a_0$

$a_0\ \$\ a\ b\ a\ a\ b_0$

$a_1\ a\ b\ a\ \$\ a\ b_1$

$a_2\ b\ a\ \$\ a\ b\ a_1$

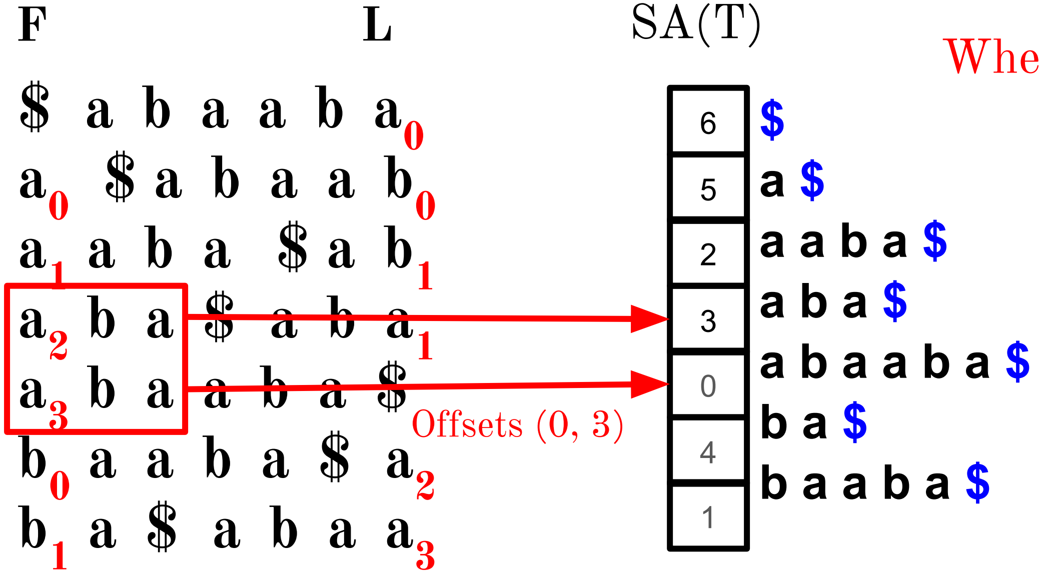
$a_3\ b\ a\ a\ b\ a\ \$$

$b_0\ a\ a\ b\ a\ \$\ a_2$

$b_1\ a\ \$\ a\ b\ a\ a_3$

Where are these?

P = aba



F L

$\$\ a\ b\ a\ a\ b\ a_0$

$a_0\ \$\ a\ b\ a\ a\ b_0$

$a_1\ a\ b\ a\ \$\ a\ b_1$

$a_2\ b\ a\ \$\ a\ b\ a_1$

$a_3\ b\ a\ a\ b\ a\ \$$

$b_0\ a\ a\ b\ a\ \$\ a_2$

$b_1\ a\ \$\ a\ b\ a\ a_3$

Offsets (0, 3)

SA(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 $

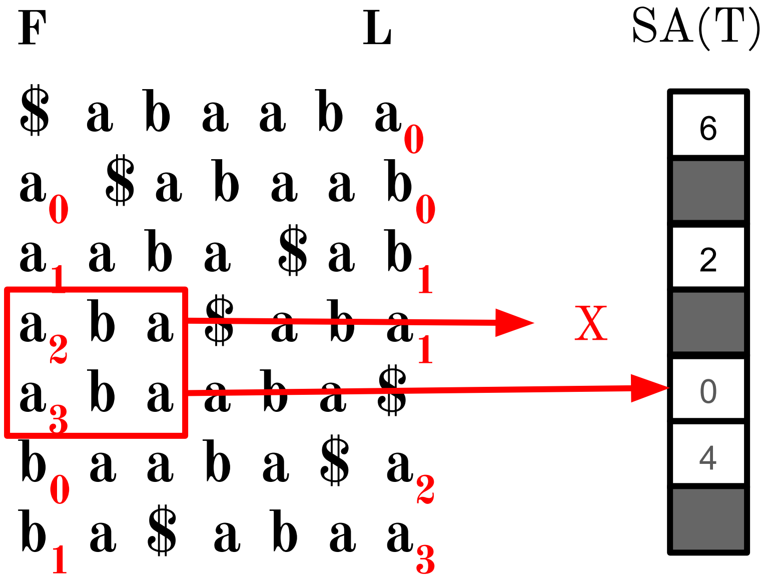
If suffix array were part of index, we could simply look up the offsets...

But, SA requires m integers...

Slide adapted from Ben Langmead

FM Index: resolving offsets

P = aba



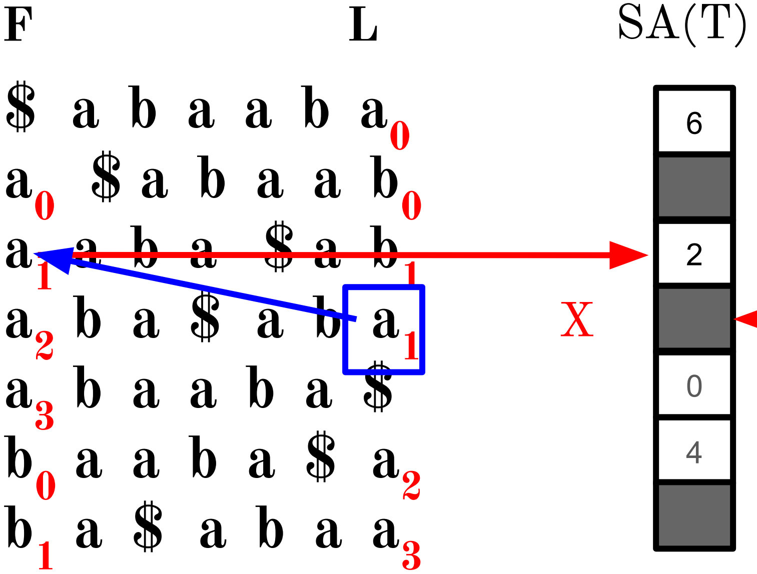
Lookup for row 4 succeeds - we kept that entry of SA

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

Slide adapted from Ben Langmead

FM Index: resolving offsets

$\mathbf{P} = \mathbf{aba}$



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 $

SA(T)

6
▨
2
▨
0
4
▨

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

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

And row 2 has a suffix array value = 2

So row 3 has suffix array value:

2 (row 2's SA val) + 1 (# steps to row 2) = 3

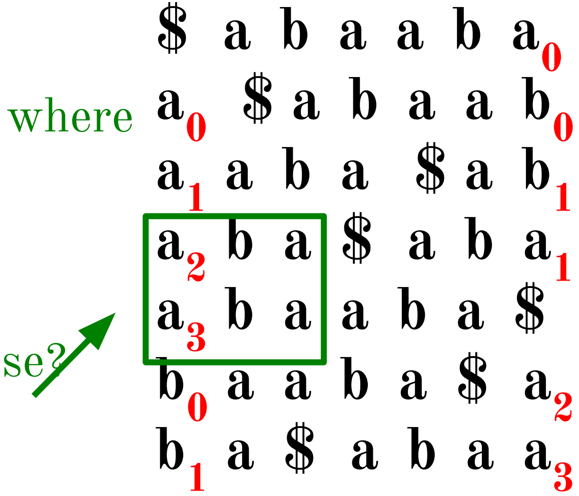
If saved SA values are O(1) positions apart in T, resolving offset is O(1) time

Slide adapted from Ben Langmead

FM Index: resolving offsets

SA sample(T)
6
▨
2
▨
0
4
▨

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



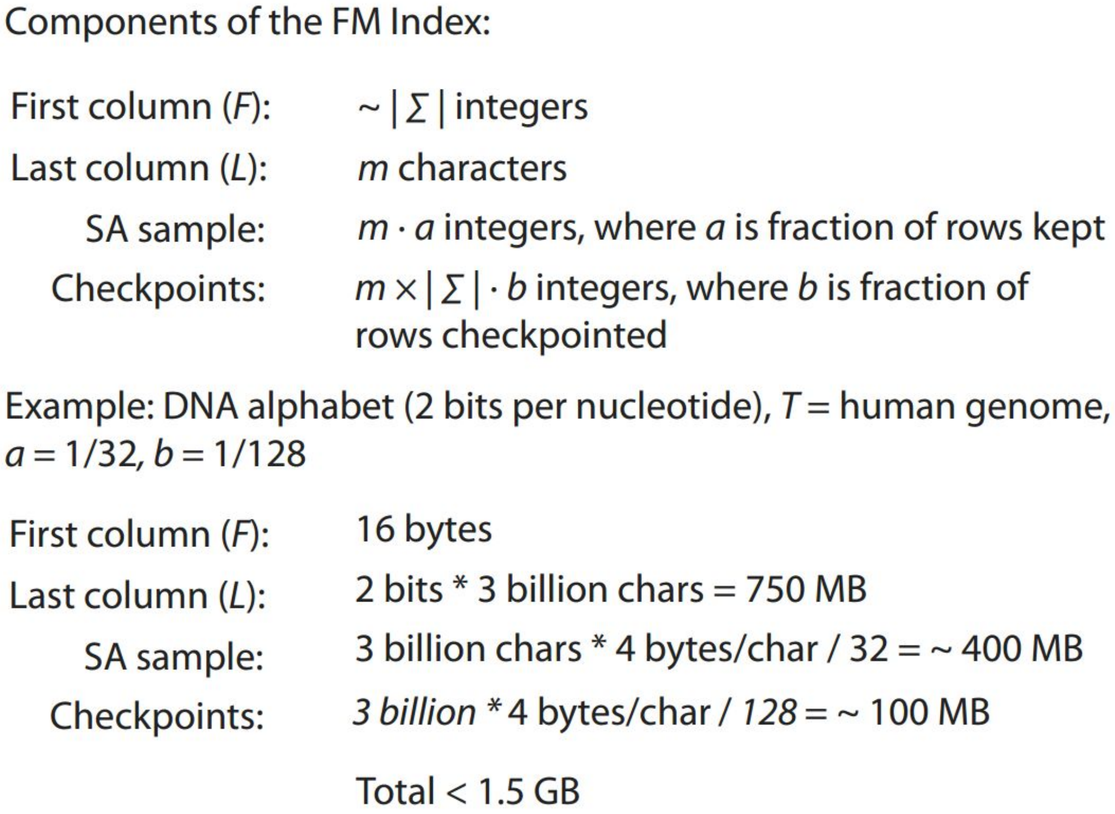
$\$$	a	b	a	a	b	a_0
a_0	$\$$	a	b	a	a	b_0
a_1	a	b	a	$\$$	a	b_1
a_2	b	a	$\$$	a	b	a_1
a_3	b	a	a	b	a	$\$$
b_0	a	a	b	a	$\$$	a_2
b_1	a	$\$$	a	b	a	a_3

Where are these?
In SA sample!

With SA sample we can do this in O(1) time per occurrence

Slide adapted from Ben Langmead

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 / *128* = ~ 100 MB

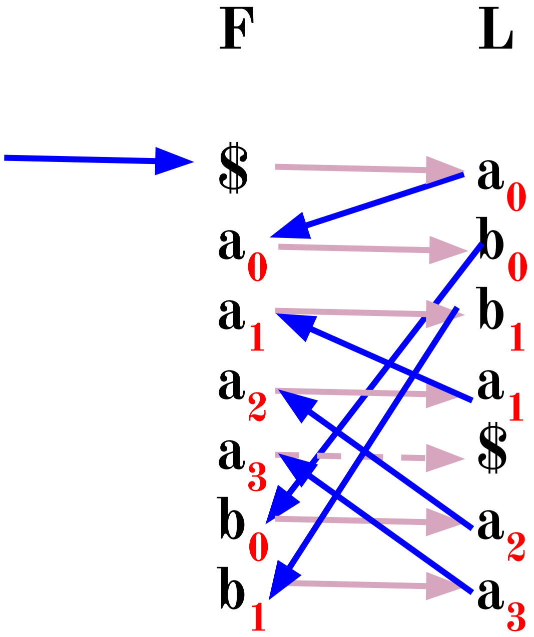
Total < 1.5 GB

Slide adapted from Ben Langmead

One more time: BWT 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
…



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

Reverse of chars we visited = a_3 b_1 a_1 a_2 b_0 a_0 $ = T

Slide adapted from Ben Langmead

One more time: BWT 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

$\$\ a\ b\ a\ a\ b\ a_0$

$a_0\ \$\ a\ b\ a\ a\ b_0$

$a_1\ a\ b\ a\ \$\ a\ b_1$

$a_2\ b\ a\ \$\ a\ b\ a_1$

$a_3\ b\ a\ a\ b\ a\ \$$

$b_0\ a\ a\ b\ a\ \$\ a_2$

$b_1\ a\ \$\ a\ b\ a\ a_3$

Look at those rows in L.

b_0, b_1 are b-s occurring just to left.

P = aba

$\$\ a\ b\ a\ a\ b\ a_0$

$a_0\ \$\ a\ b\ a\ a\ b_0$

$a_1\ a\ b\ a\ \$\ a\ b_1$

$a_2\ b\ a\ \$\ a\ b\ a_1$

$a_3\ b\ a\ a\ b\ a\ \$$

$b_0\ a\ a\ b\ a\ \$\ a_2$

$b_1\ a\ \$\ a\ b\ a\ a_3$

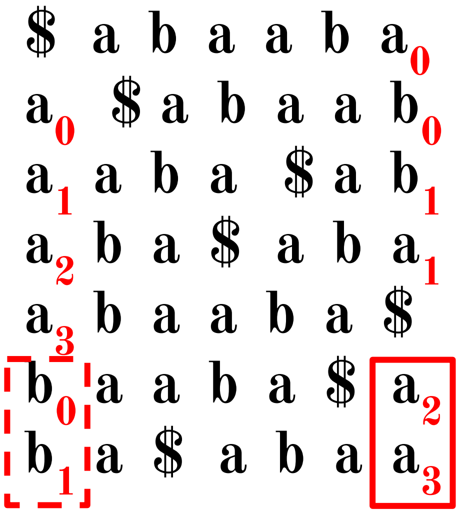
Use LF Mapping. Let new range delimit those b-s

Slide adapted from Ben Langmead

One more time: BWT FM Index - querying

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

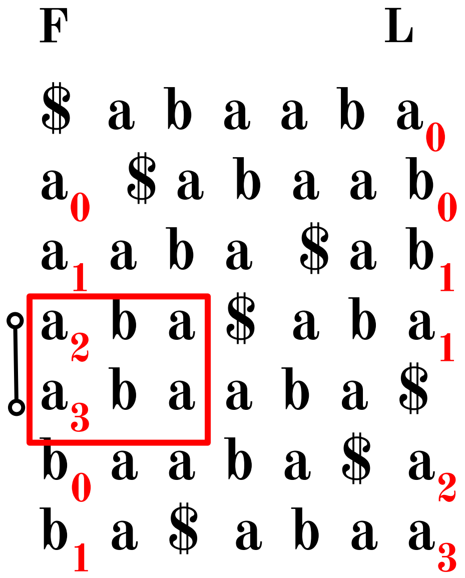
P = aba



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

Occurs just to the left

P = aba



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

Use LF mapping

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

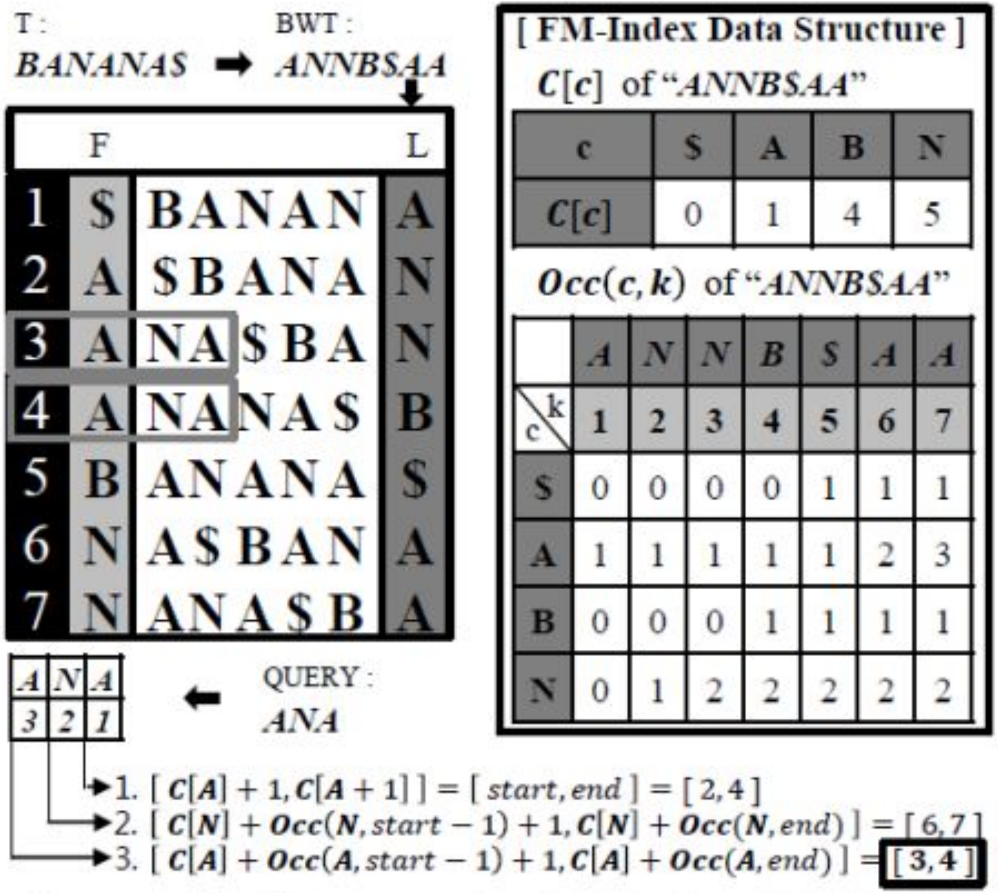
Slide adapted from Ben Langmead

FM Index

1. L = BWT(T)
2. First column (number of appearances of each character)
3. Suffix Array (or SA Sample)
4. Tally (rank, occurrences) matrix

Slide adapted from Ben Langmead

FM Index: Example



T: *BANANA$* → BWT: *ANNB$AA*

	F		L
1	$	BANAN	A
2	A	$BANA	N
3	A	NA$BA	N
4	A	NANA$	B
5	B	ANANA	$
6	N	A$BAN	A
7	N	ANA$B	A

[FM-Index Data Structure]

$C[c]$ of "*ANNB$AA*"

c	$	A	B	N
$C[c]$	0	1	4	5

$Occ(c,k)$ of "*ANNB$AA*"

	A	*N*	*N*	*B*	*$*	*A*	*A*
c \ k	1	2	3	4	5	6	7
$	0	0	0	0	1	1	1
A	1	1	1	1	1	2	3
B	0	0	0	1	1	1	1
N	0	1	2	2	2	2	2

A	*N*	*A*
3	*2*	*1*

← QUERY: *ANA*

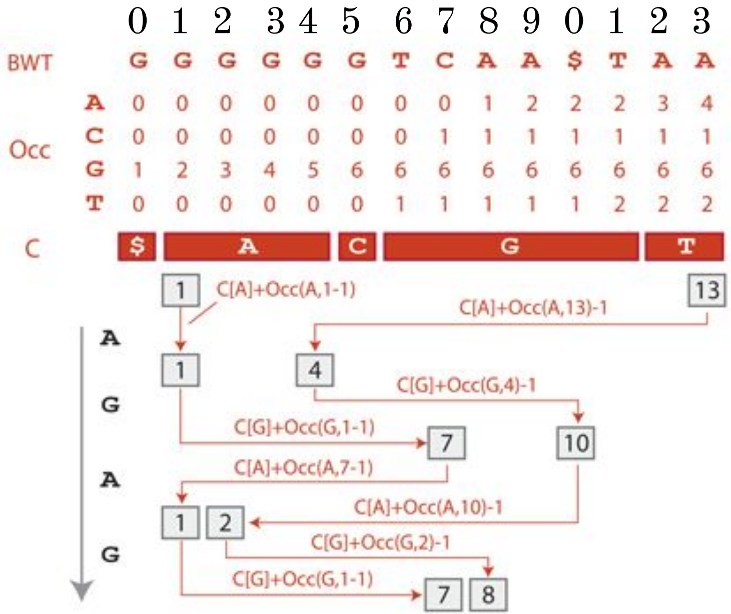
1. $[\, C[A]+1, C[A+1] \,] = [\, start, end \,] = [2,4]$
2. $[\, C[N]+Occ(N, start-1)+1, C[N]+Occ(N, end) \,] = [6,7]$
3. $[\, C[A]+Occ(A, start-1)+1, C[A]+Occ(A, end) \,] = \boxed{[3,4]}$

Fig. 3. An example of query search using BWT and FM-index for text T=*BANANA$*.The $ is 'EOF' character.

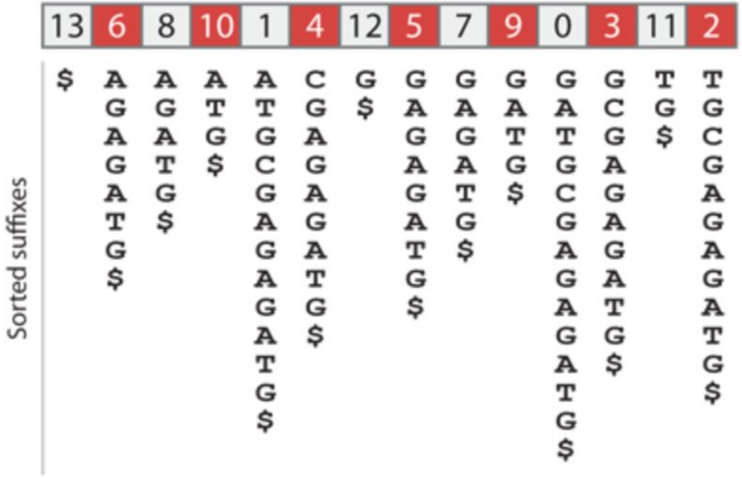
A Very Fast Algorithm for Detecting Partially Plagiarized Documents Using FM-Index

FM Index: Example

Search for:
GAGA



	0	1	2	3	4	5	6	7	8	9	0	1	2	3
BWT	G	G	G	G	G	G	T	C	A	A	$	T	A	A
Occ A	0	0	0	0	0	0	0	0	1	2	2	2	3	4
Occ C	0	0	0	0	0	0	0	1	1	1	1	1	1	1
Occ G	1	2	3	4	5	6	6	6	6	6	6	6	6	6
Occ T	0	0	0	0	0	0	1	1	1	1	1	2	2	2



13	6	8	10	1	4	12	5	7	9	0	3	11	2
$	AGAGATG$	AGATG$	ATG$	ATGCGAGAGATG$	CGAGAGATG$	G$	GAGAGATG$	GAGATG$	GATG$	GATGCGAGAGATG$	GCGAGAGATG$	TG$	TGCGAGAGATG$

Source

Usage of BWT FM index in real life?

Slide adapted from Ben Langmead

FASTQ

```
@SEQ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65

                                ...
```

A FASTQ (FQ) file normally uses four lines per sequence.

- Line 1 begins with a '@' character and is followed by a sequence identifier and an optional description.
- Line 2 is the raw sequence letters.
- Line 3 begins with a '+' character and is optionally followed by the same sequence identifier (and any description) again.
- Line 4 encodes the quality values for the sequence in Line 2, and must contain the same number of symbols as letters in the sequence.

FASTA

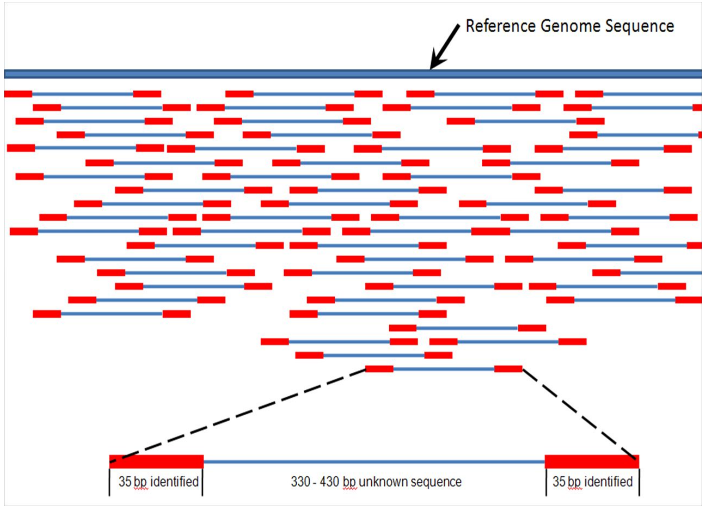
```
> CONTIG_NAME
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
GATTTGGGGTTCAAAGCAGTAATTTGGGGTTCAAAGCAGTATCGACAAATAGTAAATCCA
TTTGTTCATTCAAAGCAGTAATTTGGGGTTATTTGGGGTTCAAAGCAGTATCGATCAAAT
AGTAAATCCATTTGTTCAACTCACAGTTT
GATT
```

FASTA is used for storing the sequence of nucleotides or amino acids

BWA-MEM

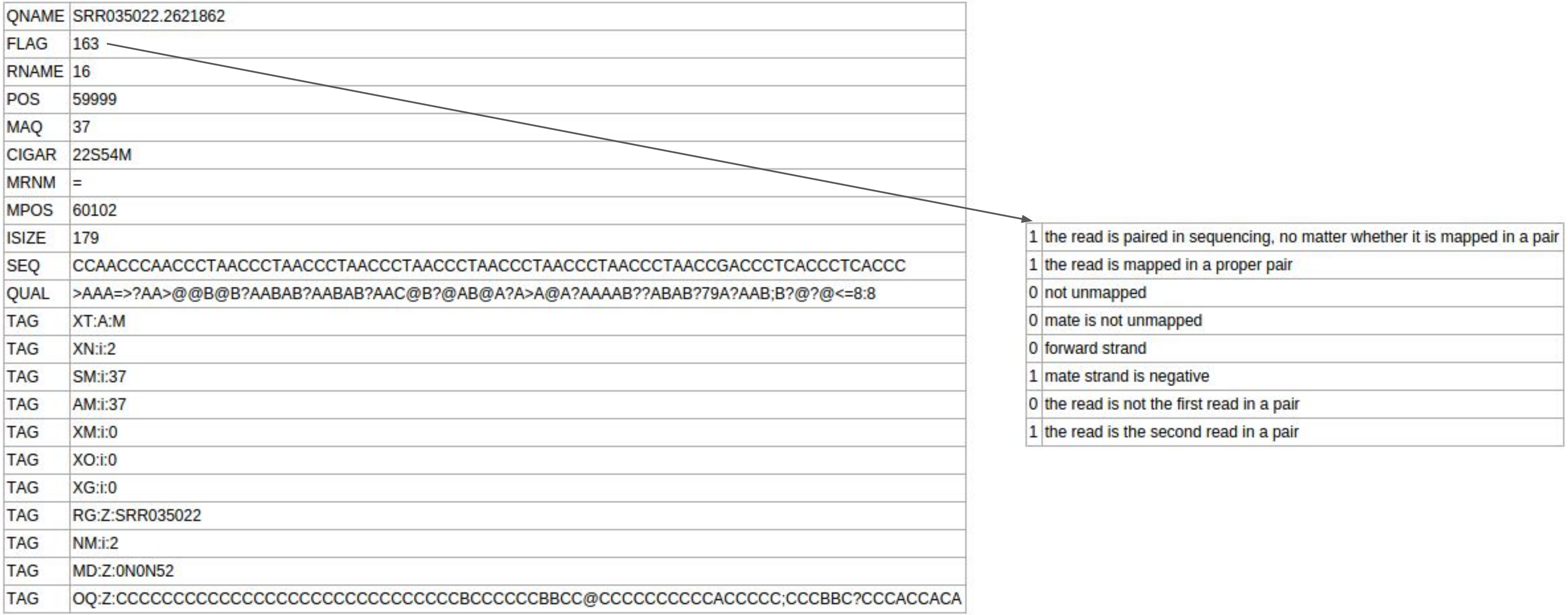
```
bwa mem ref.fa read1.fq read2.fq > aln.sam
```

- http://bio-bwa.sourceforge.net/
- Reference genome index must exist
- Paired-end reads
- Primary and secondary alignment (random)



BWA-MEM output

```
Line from SAM file:
SRR035022.2621862 163 16 59999 37 22S54M = 60102 179 CCAACCCAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCGACCCTCACCCTCACCC
>AAA=>?AA>@@B@B?AABAB?AABAB?AAC@B?@AB@A?A>A@A?AAAAB??ABAB?79A?AAB;B?@?@<=8:8 XT:A:M XN:i:2 SM:i:37 AM:i:37 XM:i:0 XO:i:0 XG:i:0 RG:Z:SRR035022 NM:i:2
MD:Z:0N0N52 OQ:Z:CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCBCCCCCCBBCC@CCCCCCCCCCACCCCC;CCCBBC?CCCACCACA@
```



QNAME	SRR035022.2621862
FLAG	163
RNAME	16
POS	59999
MAQ	37
CIGAR	22S54M
MRNM	=
MPOS	60102
ISIZE	179
SEQ	CCAACCCAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCGACCCTCACCCTCACCC
QUAL	>AAA=>?AA>@@B@B?AABAB?AABAB?AAC@B?@AB@A?A>A@A?AAAAB??ABAB?79A?AAB;B?@?@<=8:8
TAG	XT:A:M
TAG	XN:i:2
TAG	SM:i:37
TAG	AM:i:37
TAG	XM:i:0
TAG	XO:i:0
TAG	XG:i:0
TAG	RG:Z:SRR035022
TAG	NM:i:2
TAG	MD:Z:0N0N52
TAG	OQ:Z:CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCBCCCCCCBBCC@CCCCCCCCCCACCCCC;CCCBBC?CCCACCACA

1	the read is paired in sequencing, no matter whether it is mapped in a pair
1	the read is mapped in a proper pair
0	not unmapped
0	mate is not unmapped
0	forward strand
1	mate strand is negative
0	the read is not the first read in a pair
1	the read is the second read in a pair

BWA-MEM performance on real data

Total reads size [Gb]	Instance	Execution time
13.6	C3.2xlarge (8CPUs, 15GB)	2h,11min
23.8	C3.2xlarge (8CPUs, 15GB)	2h, 45min
100	C3.8xlarge (32CPUs, 60GB)	5h, 30min

References

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

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING