

22
MAR

Friday, March 22, 2019

SBG Meetup - Bioinformatics



Hosted by Marija H. and Andjelka

From Seven Bridges Meetups

Public group



Details

Hello and welcome to our first meetup in 2019 where we will cover some interesting topics in the field of bioinformatics. Feel free to join us and our guest Vladimir Perovic from Vinca Institute for a vibrant talk on Friday 22nd of March! Please read on for more information about our agenda and talkers.

Agenda:

18:00h - Prediction of protein functions and interactions using machine learning algorithms
Vladimir Perovic; Computer Scientist, Research Associate at Vinča Institute

18:40h - Personalized cancer therapy: Discovery of neoantigen cancer markers in the era of NGS data
Vladimir Kovacevic; Bioinformatics Engineer at Seven Bridges

19:20h - Networking, snacks and 🍷

<https://www.meetup.com/seven-bridges/events/259527619/>

Learn from cancer genomics data.

FASTER

The Cancer Genomics Cloud (CGC), powered by [Seven Bridges](#), is one of three [systems](#) funded by the [National Cancer Institute](#) to explore the paradigm of colocalizing massive public datasets, like The Cancer Genomics Atlas ([TCGA](#)), alongside secure and scalable computational resources to analyze them. The CGC makes more than two petabytes of multi-dimensional data available immediately to authorized researchers. You can add your own data to analyze alongside the public datasets using predefined analytical workflows or your own tools. Every execution is fully reproducible, and collaborating with your team is simple and secure.



[THE CGC HISTORY](#)



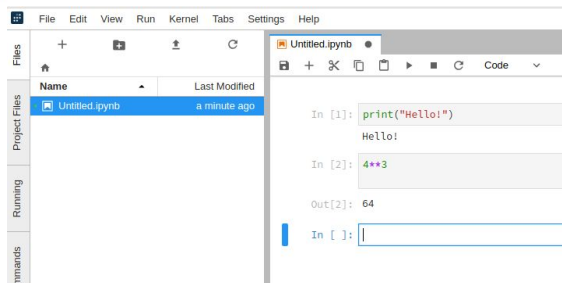
[ACCESS TCGA DATA](#)



[BRING YOUR TOOLS](#)



[BRING YOUR
PRIVATE DATA](#)



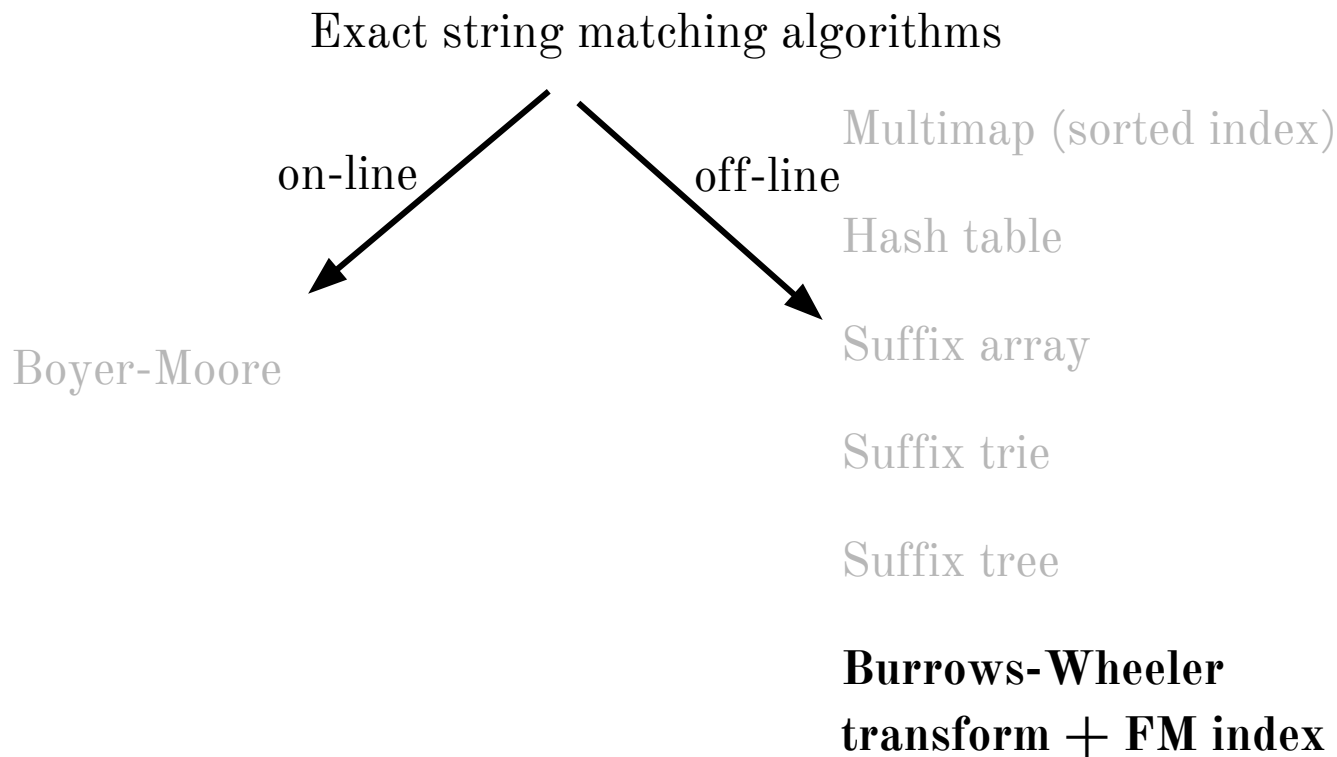
<https://transfer.sh/>

Burrows-Wheeler Transform and FM Index

—

Lesson 04

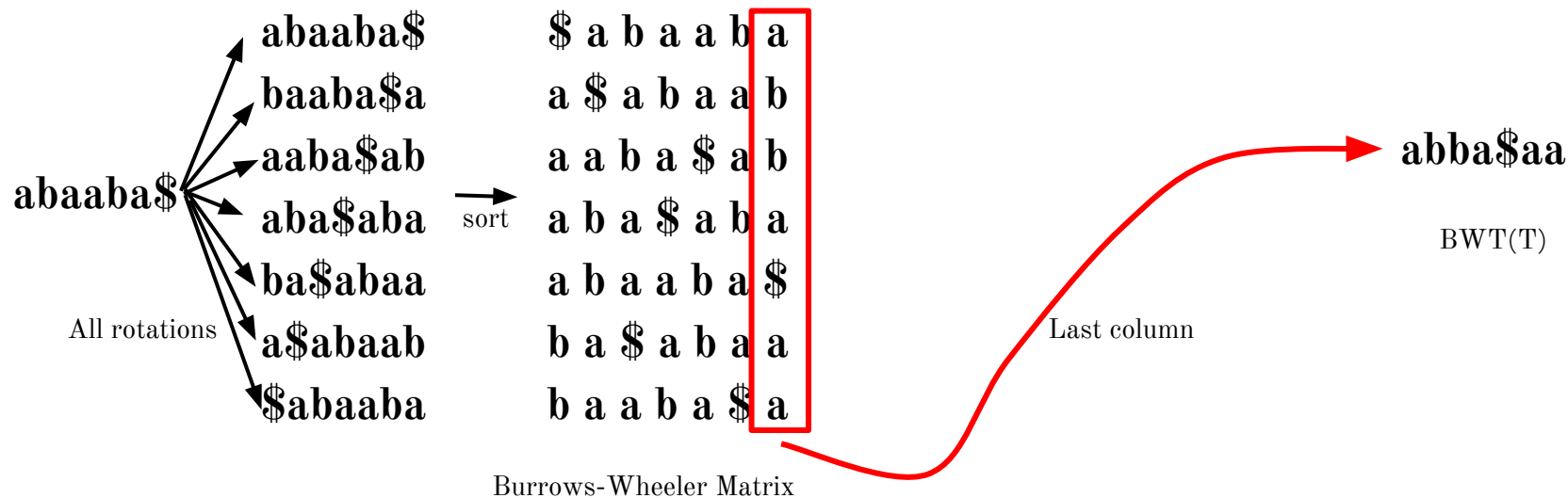
Recapitulation



Burrows-Wheeler Transform



Burrows-Wheeler Transform



How is it useful for compression?

How is it reversible?

How is it an index?

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)) ]
```

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

Take last column

```
>>> bwtViaBwm("Tomorrow_and_tomorrow_and_tomorrow$")  
'w$wwdd__nnooaaattTmmmrrooo__ooo'  
>>> bwtViaBwm("It_was_the_best_of_times_it_was_the_worst_of_times$")  
's$esttssffteww_hhmmbootttt_ii_woeaaressIi_____  
>>> bwtViaBwm('in_the_jingle_jangle_morning_Ill_come_following_you$')  
'u_gleeeengj_mhl_nnnnt$nwj__lggIolo_iiiarfcmylo_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

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-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

BWT(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)

Which structure is
very similar to BWM?

Sort order is the same whether rows are rotations or suffixes

Burrows-Wheeler Transform

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

\$ 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

BWT(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)

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

Burrows-Wheeler Transform

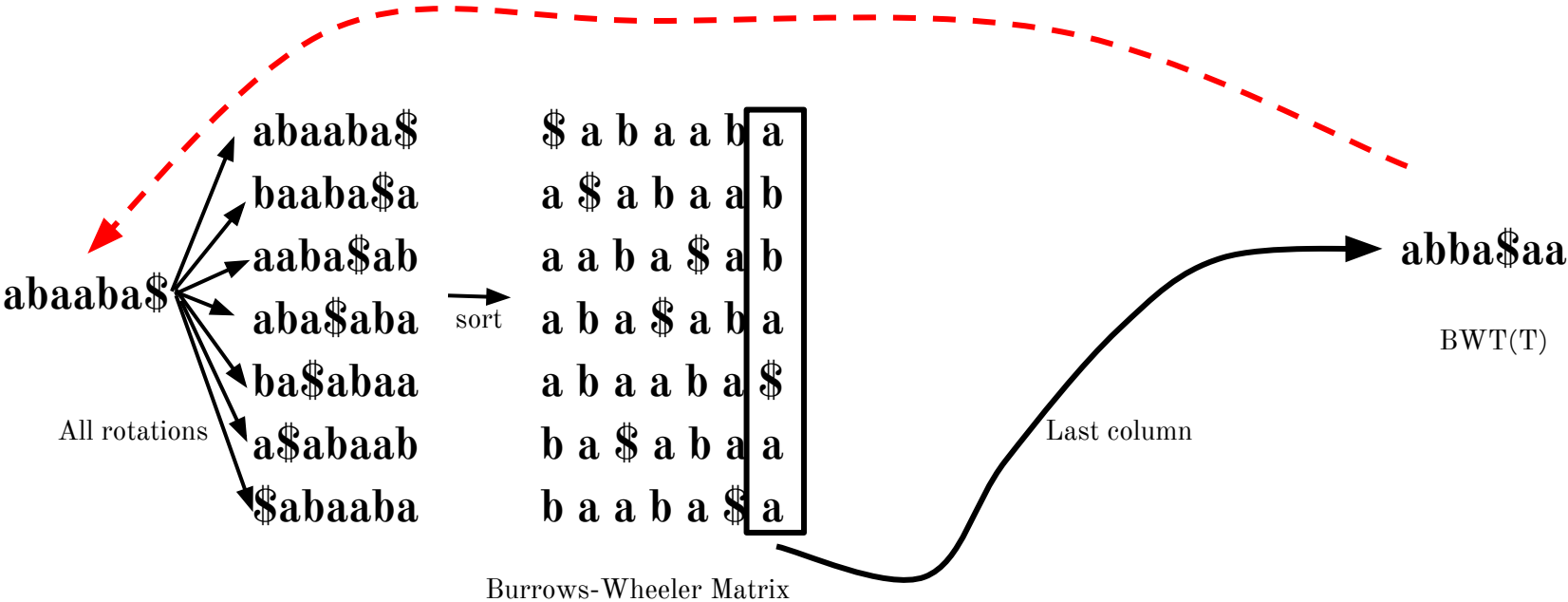
```
def suffixArray(s):  
    """ Given T return suffix array SA(T). We use Python's sorted  
        function here for simplicity, but we can do better. """  
    satups = sorted([(s[i:], i) for i in range(len(s))])  
    # Extract and return just the offsets  
    return map(lambda x: x[1], satups)  
  
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-sized 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__nnooaaattTmmrrrrrrrooo__ooo'  
>>> bwtViaSa("It_was_the_best_of_times_it_was_the_worst_of_times$")  
's$esttssffteww_hhmmbootttt_ii_woeaaressIi_____  
>>> bwtViaSa('in_the_jingle_jangle_morning_Ill_come_following_you$')  
'u_gleeeengj_mhl_nnnnt$nwj__lggIolo_iiiiarfcmylo_oo_'
```

Burrows-Wheeler Transform



How to reverse the BWT?

BWM has a key property called the LF Mapping

Burrows-Wheeler Transform: T-ranking

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

a₀ **b**₀ **a**₁ **a**₂ **b**₁ **a**₃ \$

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

Burrows-Wheeler Transform: T-ranking

BWT 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	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

“a” 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: T-ranking

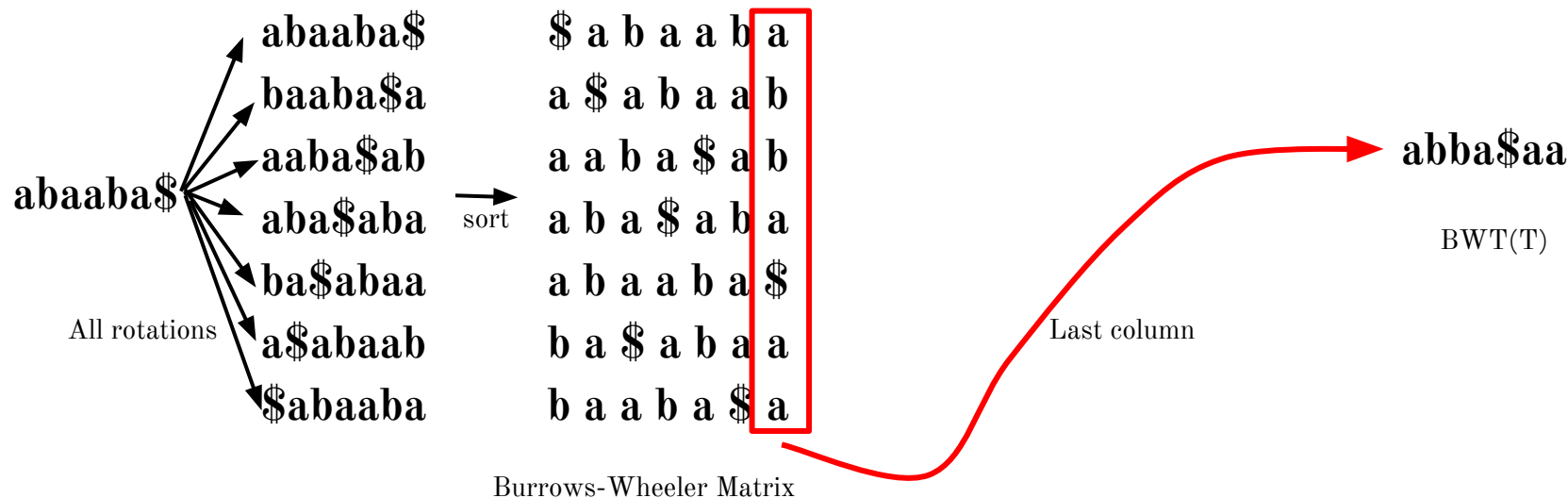
BWT with T-ranking:

	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

Same is with “b”

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

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

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	₄
a	₂	b	a	\$	a	b	a	₁
a	₀	b	a	a	b	a	\$	₁
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	\$	₁
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

Burrows-Wheeler Transform: LF Mapping

BWM with **B-ranking**:

\$	a	b	a	a	b	a	0
a	\$	a	b	a	a	b	0
a	a	b	a	\$	a	b	1
a	b	a	\$	a	b	a	1
a	b	a	a	b	a	\$	3
b	a	a	b	a	\$	a	2
b	a	\$	a	b	a	a	3

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)

Burrows-Wheeler Transform

	F	L	
	\$	a ₀	
	a ₀	b ₀	
	a ₁	b ₁	← Which BWM row begins with b₁ ?
	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 b₀ (1 row)
row 6 →	b ₁	a ₃	

Answer: row 6

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 $1 + 300 + 400 + 100 = \mathbf{row\ 801}$

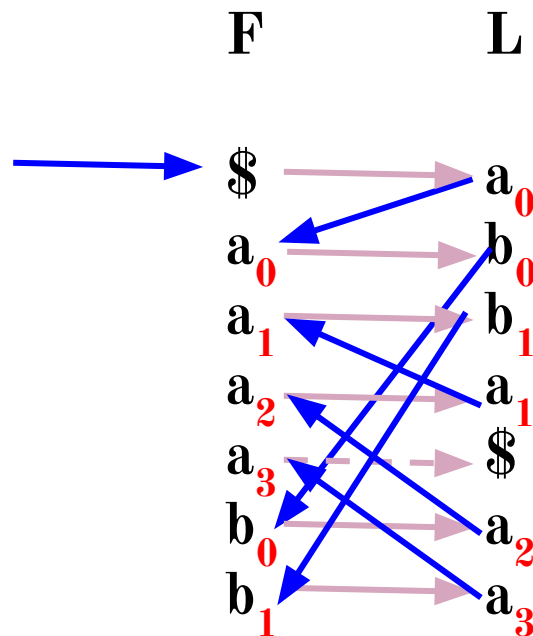
Burrows-Wheeler Transform: reversing

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

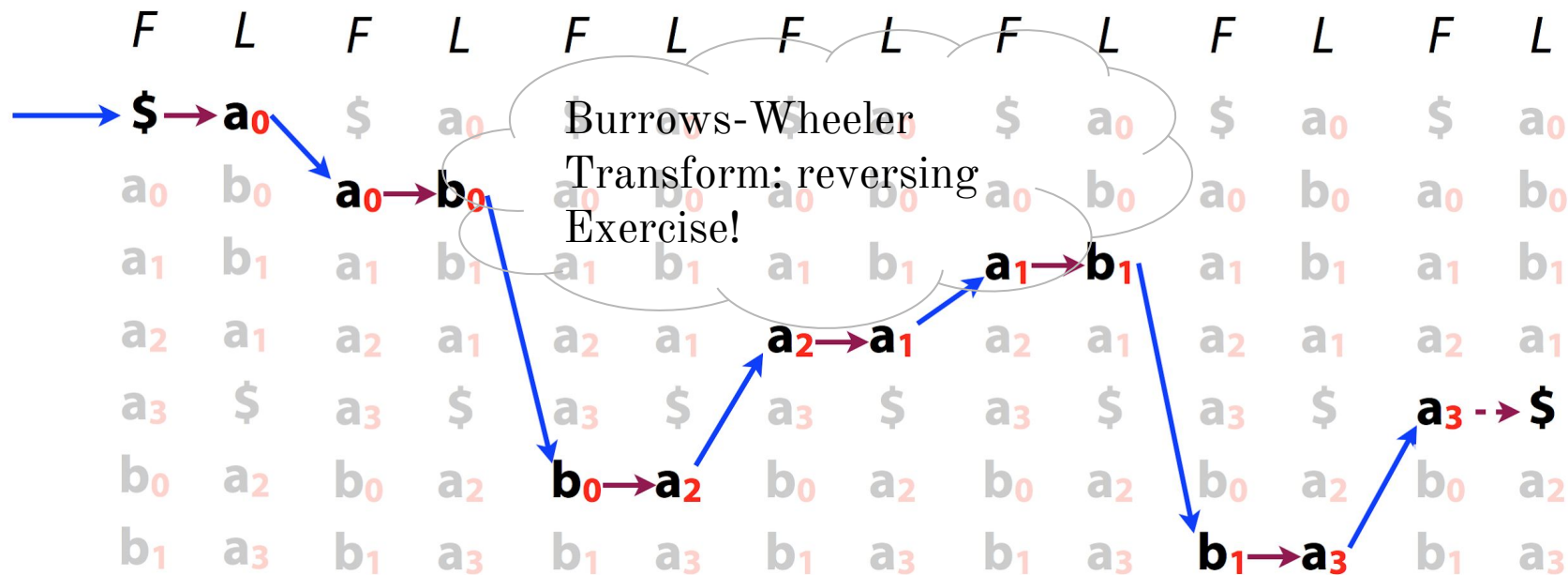
...



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

Burrows-Wheeler Transform: reversing

Another way of visualizing Reverse BWT(T)



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?

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

\$ 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 \$

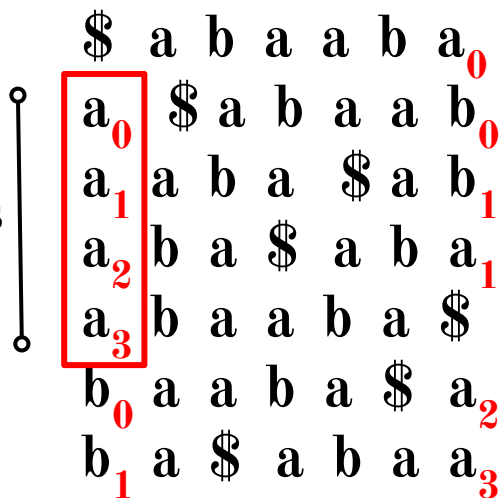
We don't have these columns; binary search isn't possible

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

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

P = ab**a**

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

$P = ab\mathbf{a}$

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

Look at those rows in L.

$\mathbf{b}_0, \mathbf{b}_1$ are b-s occurring just to left.

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

$P = a\mathbf{b}a$

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

FM Index: querying

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

$P = \text{a} \textcolor{red}{b} \textcolor{red}{a}$

\$	a	b	a	a	b	a	₀
a	₀	\$	a	b	a	a	b
a	₁	a	b	a	\$	a	b
a	₂	b	a	\$	a	b	a
a	₃	b	a	a	b	a	\$
<div style="border: 1px solid red; padding: 2px;">b</div>	₀	a	a	b	a	\$	<div style="border: 1px solid red; padding: 2px;">a</div>
<div style="border: 1px solid red; padding: 2px;">b</div>	₁	a	\$	a	b	a	<div style="border: 1px solid red; padding: 2px;">a</div>

Occurs just to the left

$P = \textcolor{red}{a} \textcolor{red}{b} \textcolor{red}{a}$

F							L
\$	a	b	a	a	b	a	₀
a	₀	\$	a	b	a	a	b
a	₁	a	b	a	\$	a	b
<div style="border: 1px solid red; padding: 2px;">a</div>	₂	b	a	\$	a	b	a
<div style="border: 1px solid red; padding: 2px;">a</div>	₃	b	a	a	b	a	\$
b	₀	a	a	b	a	\$	a
b	₁	a	\$	a	b	a	a

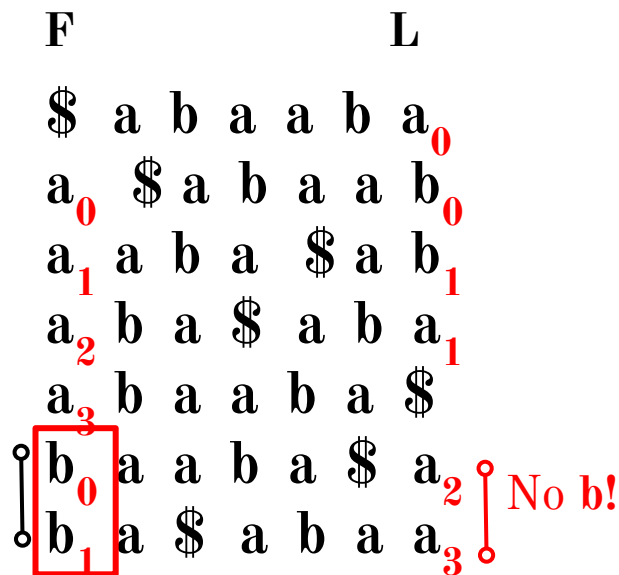
Use LF mapping

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

FM Index: querying

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

P = bba



Use LF mapping

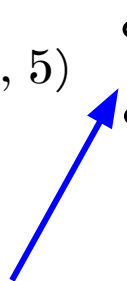
FM Index: querying

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

[3, 5)

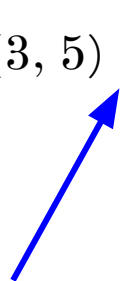


Where are these?

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 \$

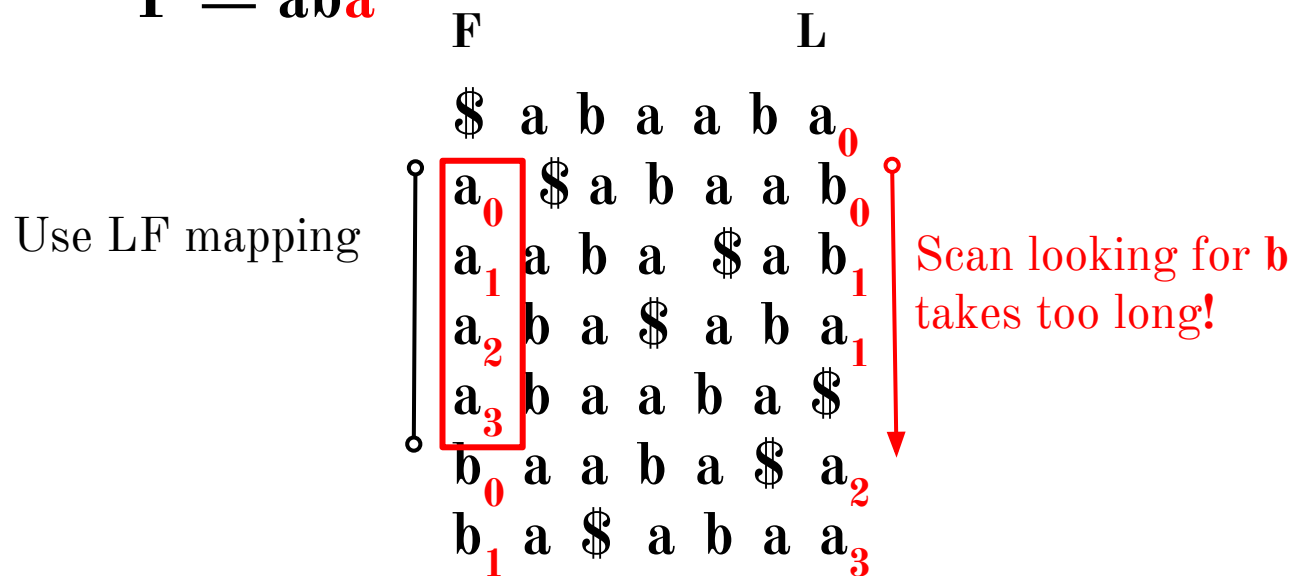
[3, 5)



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

FM Index: querying

$P = ab\mathbf{a}$



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

m integers

(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
```

(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?

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	<table><tr><td>1</td><td>0</td></tr><tr><td>1</td><td>1</td></tr><tr><td>1</td><td>2</td></tr><tr><td>2</td><td>2</td></tr><tr><td>2</td><td>2</td></tr><tr><td>3</td><td>2</td></tr><tr><td>4</td><td>2</td></tr></table>	1	0	1	1	1	2	2	2	2	2	3	2	4	2	
1	0																
1	1																
1	2																
2	2																
2	2																
3	2																
4	2																
a ₀	b ₀																
a ₁	b ₁																
a ₂	a ₁																
a ₃	\$																
b ₀	a ₂																
b ₁	a ₃																

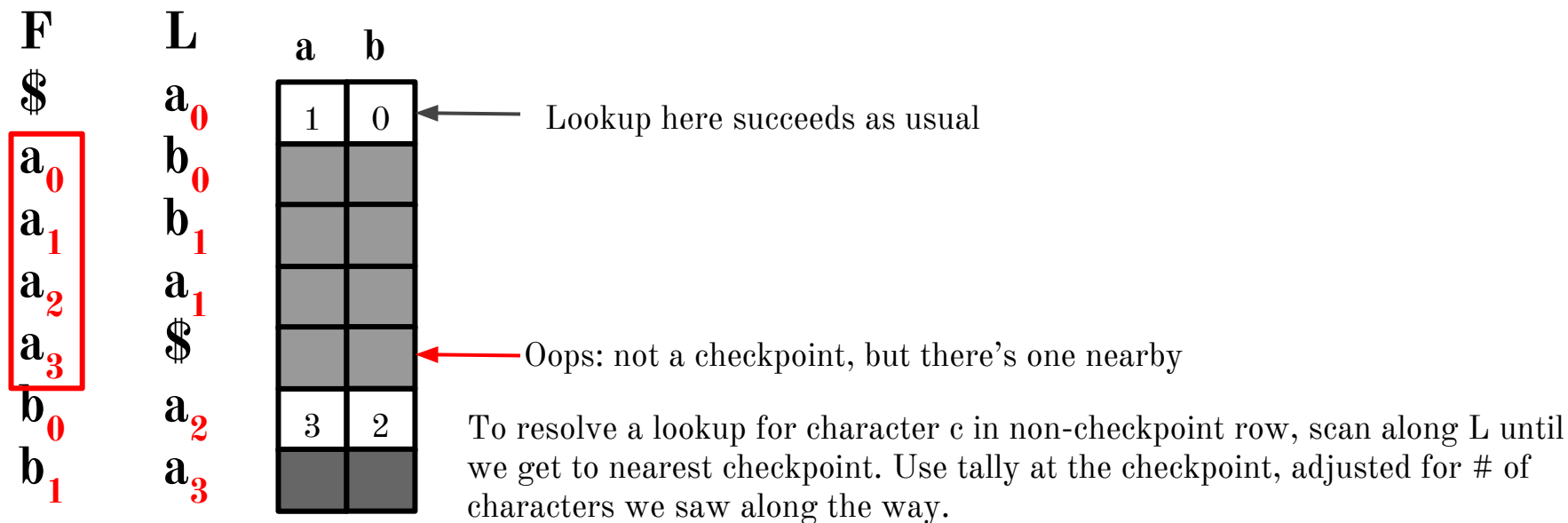
We infer b_0 and b_1
appear in L in this
range:

$$\text{Tally}(b, i) - \text{Tally}(b, j) = 2$$

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

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.



FM Index: fast rank calculations

What's my rank?

$$482 + 2 - 1 = 483$$

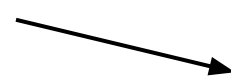
checkpoint + as along the way - tally → rank

What's my rank?

$$439 - 2 - 1 = 436$$

checkpoint + as along the way - tally → rank

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

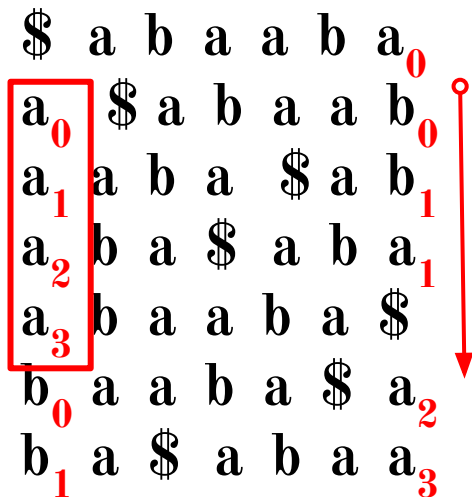


Assuming checkpoints are spaced $O(1)$ distance apart, lookups are $O(1)$

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
```

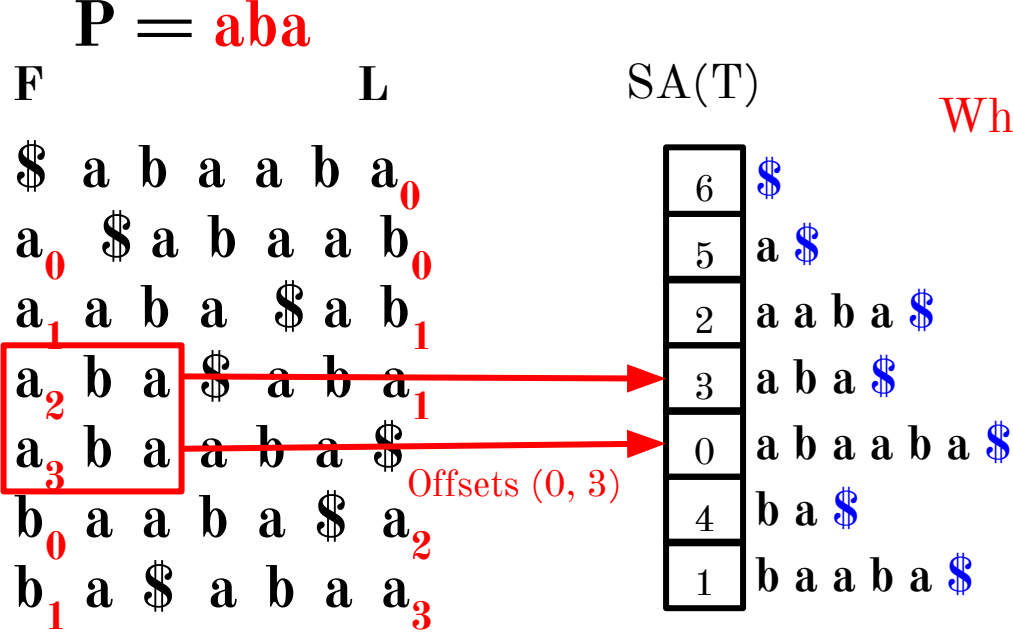


m integers

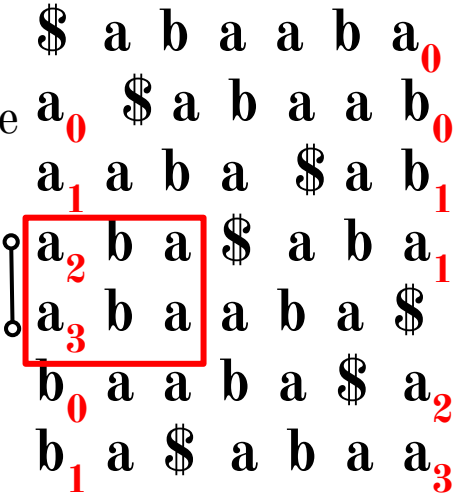
With checkpoints, we greatly reduce # integers needed for ranks - but it's still $O(m)$ space - there's literature on how to improve this space bound

FM Index: Not yet solved!

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



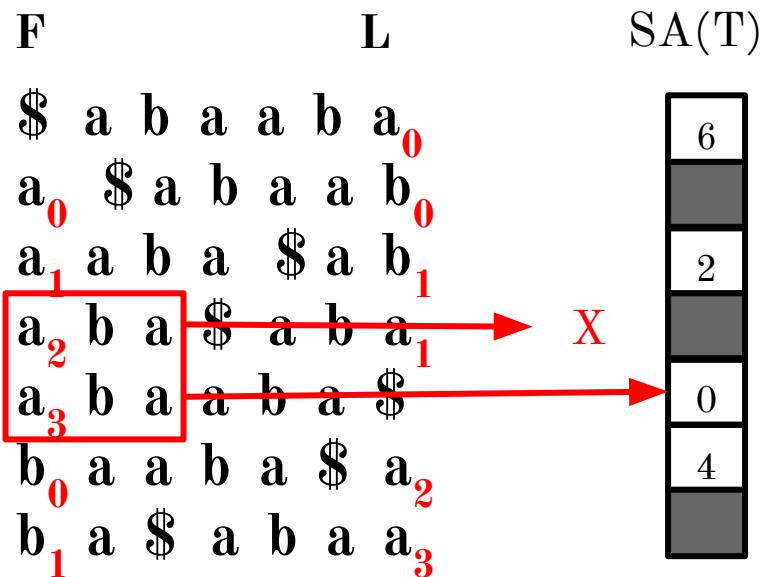
Where are these?



If suffix array were part of index,
we could simply look up the
offsets...
But, SA requires
m integers...

FM Index: resolving offsets

$P = \text{aba}$

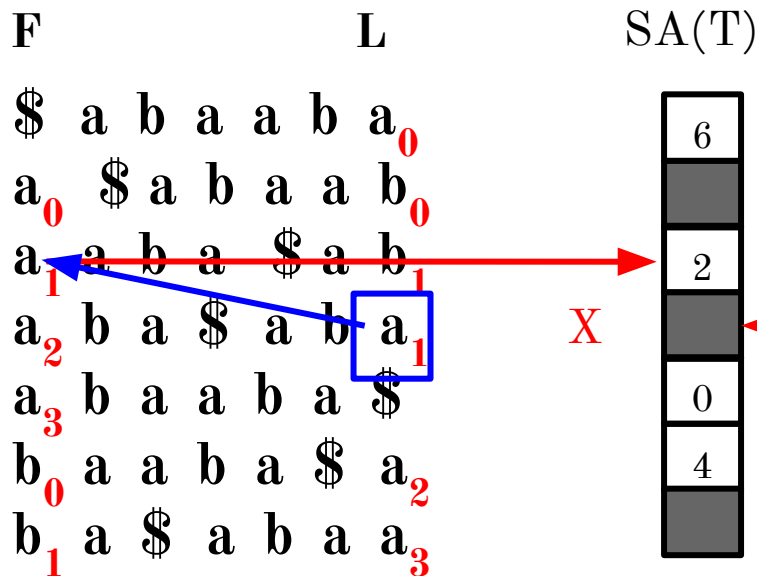


Lookup for row 4 succeeds - we kept that entry of SA

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

FM Index: resolving offsets

$P = \text{aba}$



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:

$2 \text{ (row 2's SA val)} + 1 \text{ (\# steps to row 2)} = \mathbf{3}$

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

FM Index: resolving offsets

SA sample(T)

6
2
0
4

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

Where are these?
In SA sample!

\$	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 = \sim 400 MB

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

Total < 1.5 GB

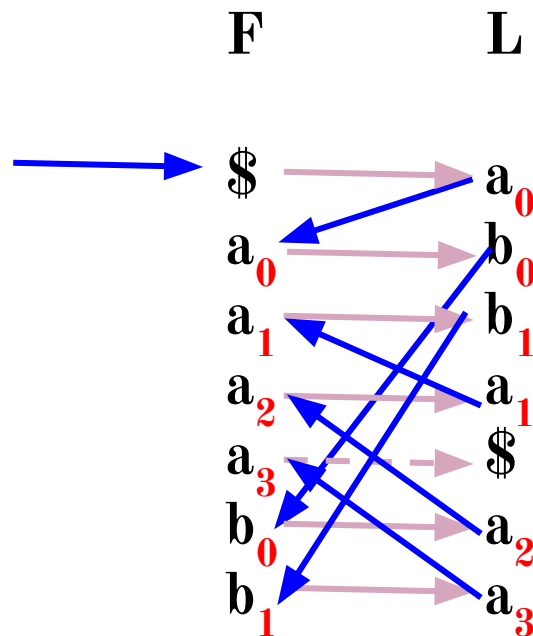
One more time: BWT reversing

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

...



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

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 = ab**a**

\$	a	b	a	a	b	a	₀
a ₀	\$	a	b	a	a	b ₀	0
a ₁	a	b	a	\$	a	b ₁	1
a ₂	b	a	\$	a	b	a ₁	1
a ₃	b	a	a	b	a	\$	
b ₀	a	a	b	a	\$	a ₂	2
b ₁	a	\$	a	b	a	a ₃	3

Look at those rows in L.

b₀, **b**₁ are b-s occurring just to left.

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

P = a**b****a**

\$	a	b	a	a	b	a	₀
a ₀	\$	a	b	a	a	b ₀	0
a ₁	a	b	a	\$	a	b ₁	1
a ₂	b	a	\$	a	b	a ₁	1
a ₃	b	a	a	b	a	\$	
b ₀	a	a	b	a	\$	a ₂	2
b ₁	a	\$	a	b	a	a ₃	3

One more time: BWT FM Index - querying

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

$P = \text{a} \textcolor{red}{ba}$

\$	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	\$	
<div style="border: 1px dashed red; padding: 2px;">b</div>	₀	a	a	b	a	\$	<div style="border: 1px solid red; padding: 2px;">a</div>	₂
<div style="border: 1px dashed red; padding: 2px;">b</div>	₁	a	\$	a	b	a	<div style="border: 1px solid red; padding: 2px;">a</div>	₃

Occurs just to the left

$P = \textcolor{red}{aba}$

	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
<div style="border: 1px solid red; padding: 2px;">a</div>	₂	b	a	\$	a	b	a	₁
<div style="border: 1px solid red; padding: 2px;">a</div>	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

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

FM Index: Example

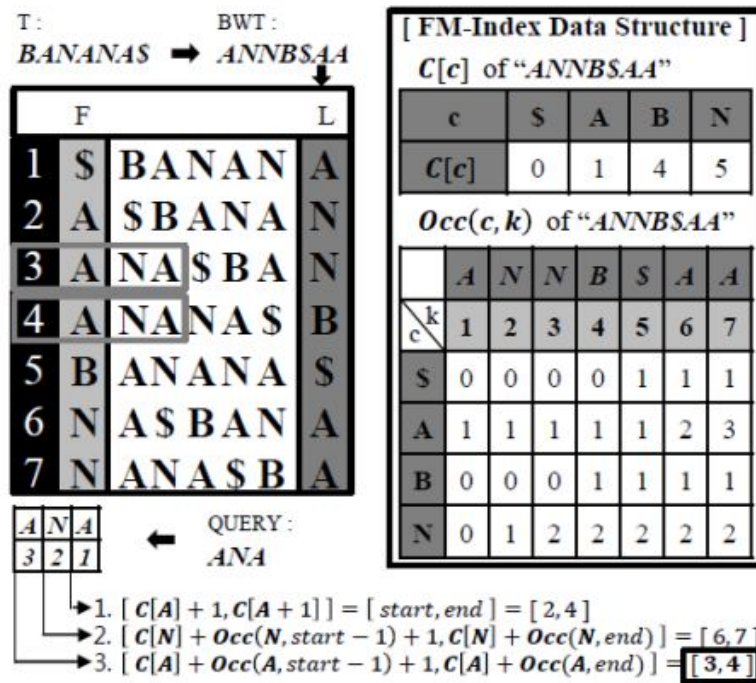
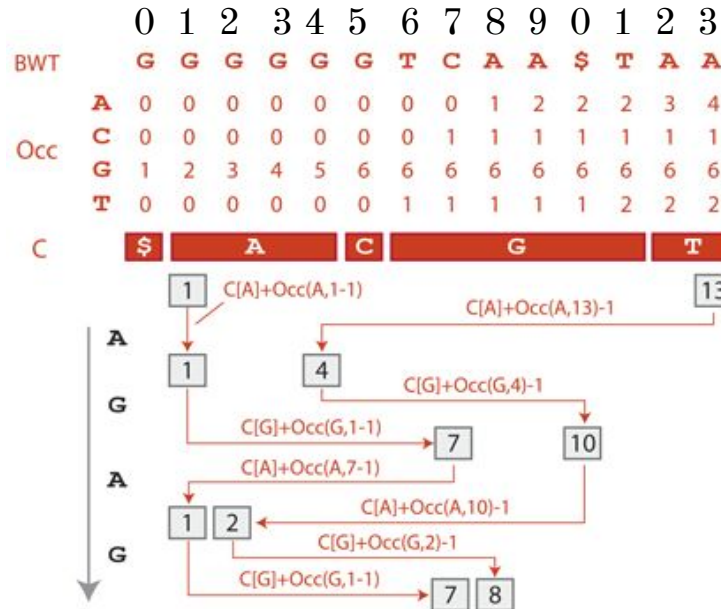


Fig. 3. An example of query search using BWT and FM-index for text T=BANANAS\$. The \$ is 'EOF' character.

FM Index: Example

Search for:
GAGA



13	6	8	10	1	4	12	5	7	9	0	3	11	2
\$	A	A	A	A	C	G	G	G	G	G	T	T	
	G	G	T	T	G	\$	A	A	A	A	G	G	
	A	A	\$	C	A		G	G	T	T	\$	C	
	G	T		G	G		A	A	G	G	A	A	
	\$	\$		A	A		T	T	A	A	T	T	
				G	G				G	G		G	
				A	A				A	A		A	
				T	T				G	G		T	
				\$	\$				\$	\$		\$	

Sorted suffixes

Usage of BWT FM index in real life?

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
TTTGTTCAATTCAAAGCAGTAATTTGGGGTTATTTGGGGTTCAAAGCAGTATCGATCAAAT
AGTAAATCCATTTGTTCAACTCACAGTTT
GATT
```

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

What is CIGAR string?

- Describes similarity between sequences

RefPos:	1	2	3	4	5	6	7		8	9	10	11	12	13	14	15	16	17	18	19
Reference:	C	C	A	T	A	C	T		G	A	A	C	T	G	A	C	T	A	A	C
Read:					A	C	T	A	G	A	A		T	G	G	C	T			

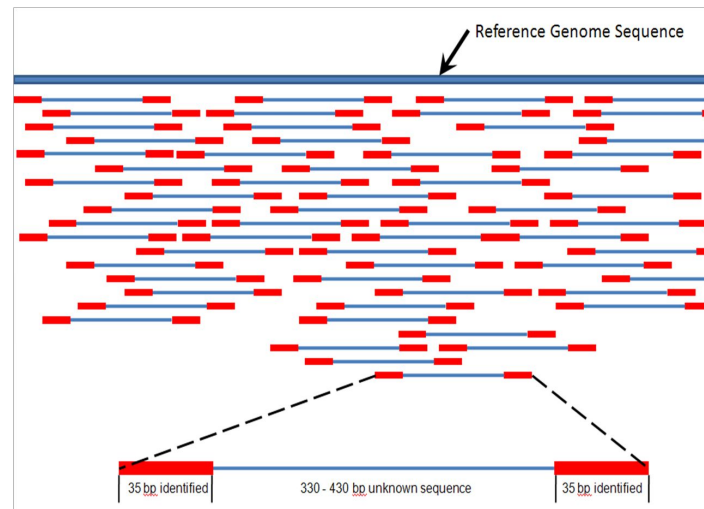
POS: 5
CIGAR: 3M1I3M1D5M

D - delete, I - insert, M - match/mismatch, H - hard clip,
S - soft clip, N - skipped from the sequence, P - padding silent

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)

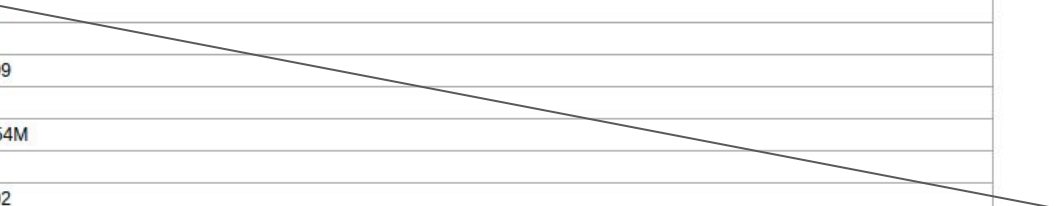


BWA-MEM output

Line from SAM file:

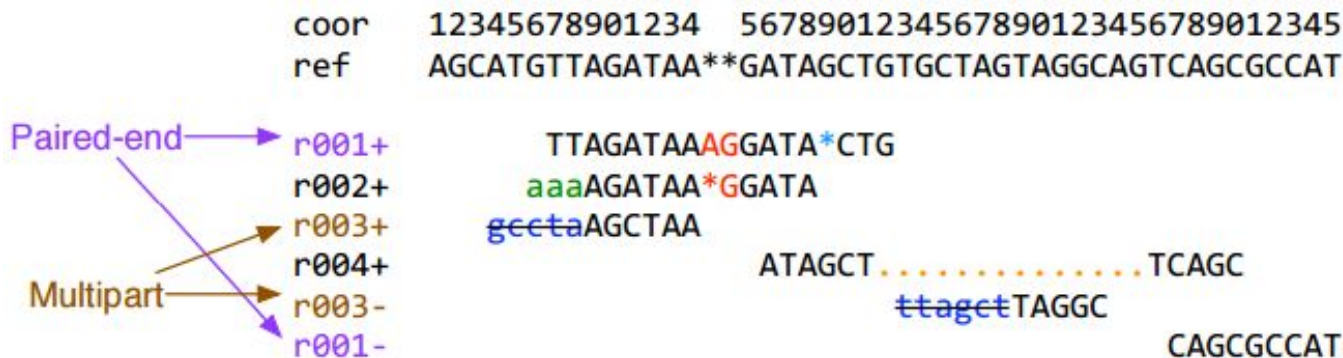
```
SRR035022.2621862 163 16 59999 37 22S54M = 60102 179 CCAACCCAACCCCTAACCCCTAACCCCTAACCCCTAACCCCTAACCCCTAACCCCTAACCGACCCCTACCCCTACCC
>AAA=>?AA>@B@B?AABAB?AABAB?AAC@B?@AB@A?A>A@A?AAAAB??ABAB?79A?AAB;B?@?@<=8:8 XT:A:M XN:i:2 SM:i:37 AM:i:37 XM:i:0 XO:i:0 XG:i:0 RG:Z:SRR035022 NM:i:2
MD:Z:0N0N52 OQ:Z:CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCBCCCCCBCC@CCCCCCCCCACCACC;C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

QNAME	SRR035022.2621862
FLAG	163
RNAME	16
POS	59999
MAQ	37
CIGAR	22S54M
MRNM	=
MPOS	60102
ISIZE	179
SEQ	CCAACCCAACCCCTAACCCCTAACCCCTAACCCCTAACCCCTAACCCCTAACCCCTAACCGACCCCTACCCCTACCC
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:CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCBCCCCCBCC@CCCCCCCCCACCACC;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 aligned reads



Ins & padding
Soft clipping
Splicing
Hard clipping

@SQ SN:ref LN:45

r001	163	ref	7	30	8M2I4M1D3M	=	37	39	TTAGATAAAGGATACTA	*
r002	0	ref	9	30	3S6M1P1I4M	*	0	0	AAAAGATAAGGATA	*
r003	0	ref	9	30	5H6M	*	0	0	AGCTAA	NM:i:1
r004	0	ref	16	30	6M14N5M	*	0	0	ATAGCTTCAGC	*
r003	16	ref	29	30	6H5M	*	0	0	TAGGC	NM:i:0
r001	83	ref	37	30	9M	=	7	-39	CAGCGCCAT	*

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

