

String Algorithms and Data Structures

Burrows-Wheeler Transform

CS 199-225

March 23, 2026

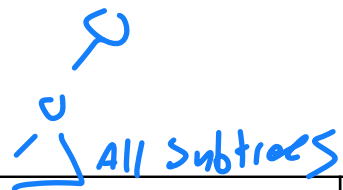
Brad Solomon





UNIVERSITY OF
ILLINOIS
URBANA - CHAMPAIGN

Department of Computer Science

Exact pattern matching *w/ indexing*


 All subtrees

start → 
 end → 

	<u>Suffix tree</u>	Suffix array
<u>Time: Does P occur?</u>	$O(P)$	$O(P \cdot \log T)$
Time: Report k locations of P	$O(P + k)$	$O(P \log T + k)$
Space	$O(n)$	$O(m)$

$m = |T|$, $n = |P|$, $k = \#$ occurrences of P in T

Exact pattern matching *w/ indexing*

	Suffix tree	Suffix array
Time: Does P occur?	$O(n)$	$O(n \log m)$
Time: Report k locations of P	$O(n + k)$	$O(n \log m + k)$
Space	$O(m)$	$O(m)$

**Suffix array
(Not covered)**

$O(n + \log m)$

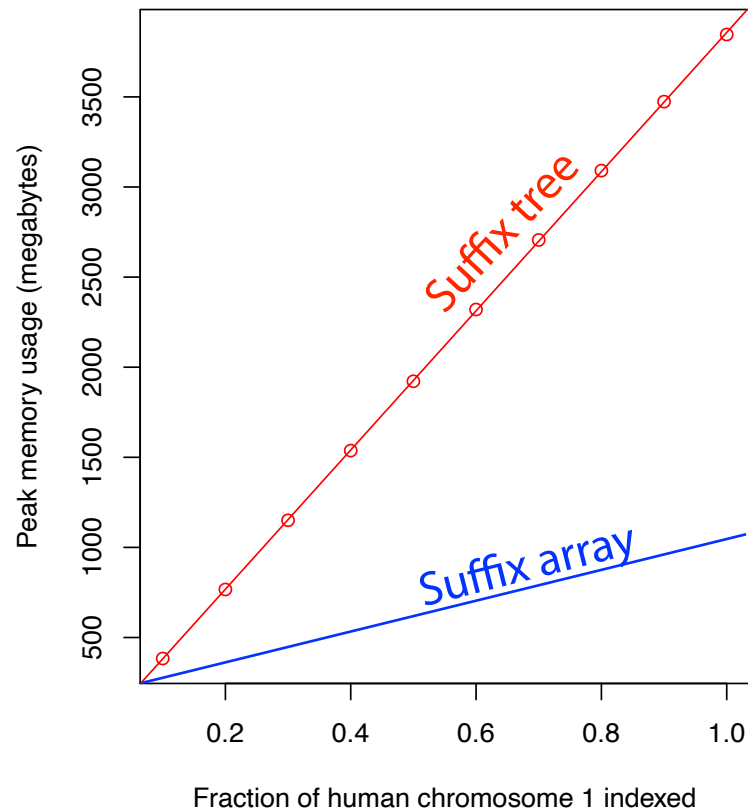
$O(n + \log m)$ *

$m = |T|$, $n = |P|$, $k = \#$ occurrences of P in T

All items match $\rightarrow k$ is $O(m)$

Suffix tree vs suffix array: size

The suffix array has a smaller constant factor than the tree



Suffix tree: ~16 bytes per character

Suffix array: ~4 bytes per character

Raw text: 2 bits per character

Exact pattern matching *w/ indexing*

The basis of the FM index is a *transformation*

T : B A N A N A \$




BWT(T): A N N B \$ A A

This transformation will frequently place characters together

As we explore this transformation, consider how and why!

Burrows-Wheeler Transform

Reversible permutation of the characters of a string


T

B A N A N A \$



BWT(T)

A N N B \$ A A

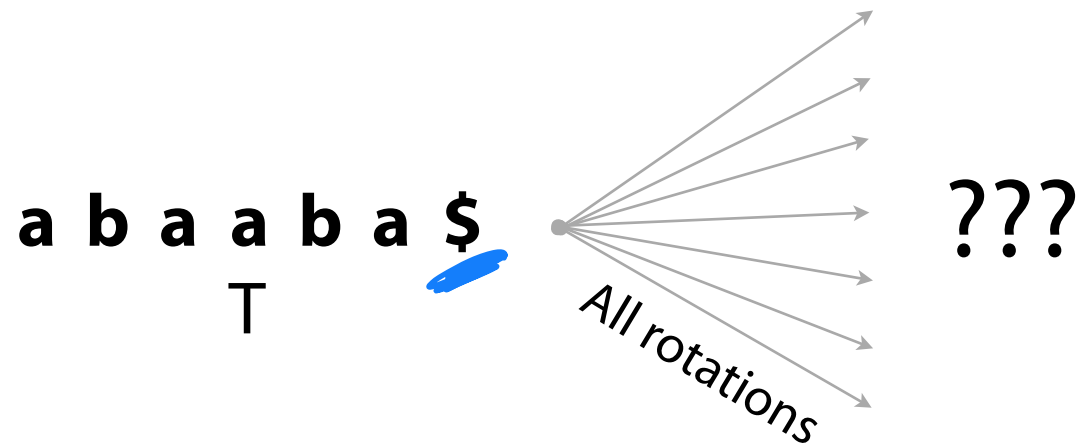
1) How to encode?

2) How to decode?

3) How is it useful for search?

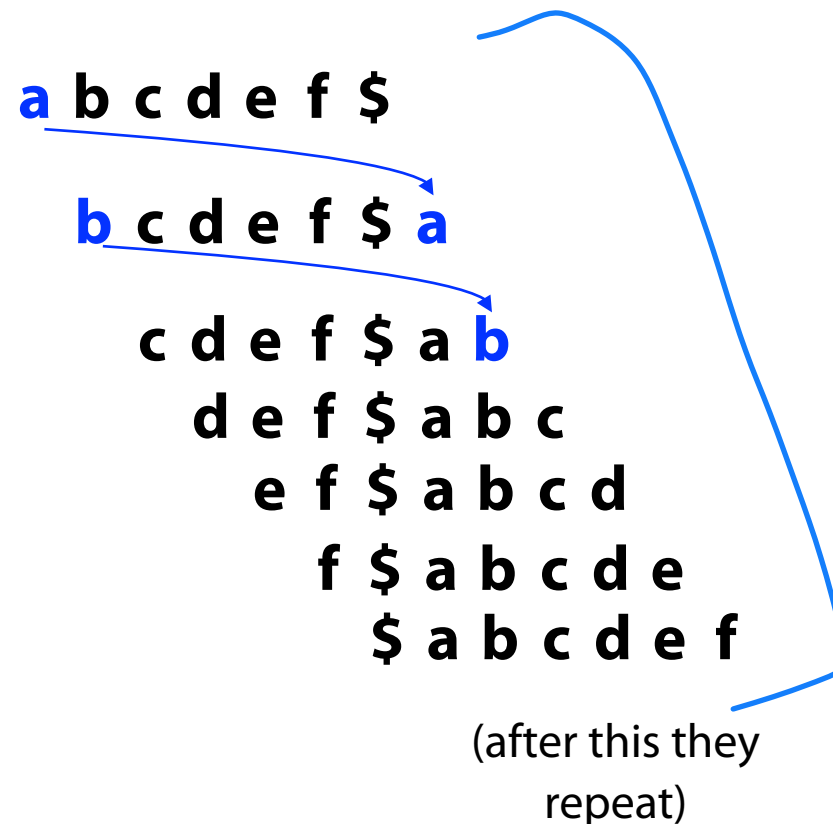
Burrows-Wheeler Transform

1) Build all **text rotations** of the input string



Text rotations

A string is a 'rotation' of another string if it can be reached by wrap-around shifting the characters



Text Rotations



Join Code: 225

A string is a 'rotation' of another string if it can be reached by wrap-around shifting the characters

Which of these are rotations of 'ABCD'?

A) BCDA

B) BACD

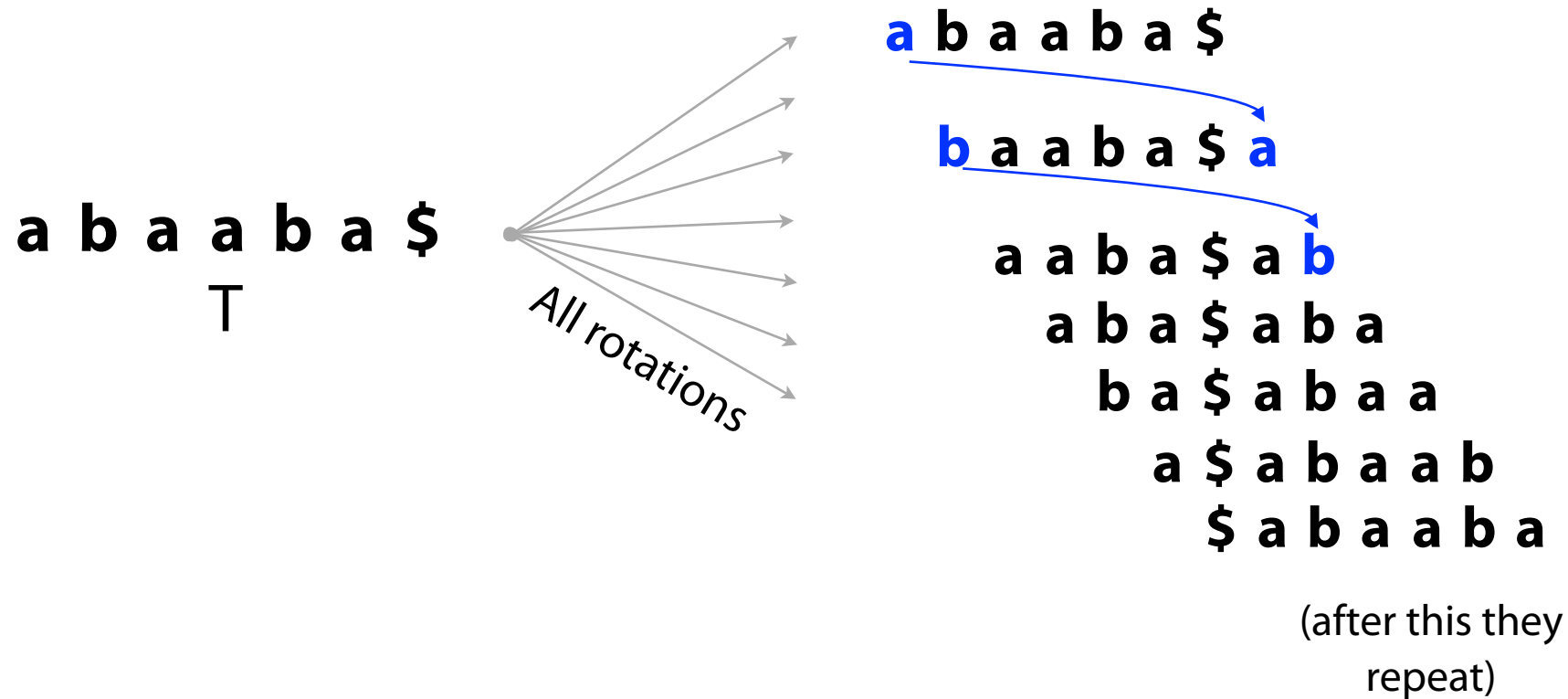
C) DCAB

D) CDAB

BCDA
CDAB
DABC

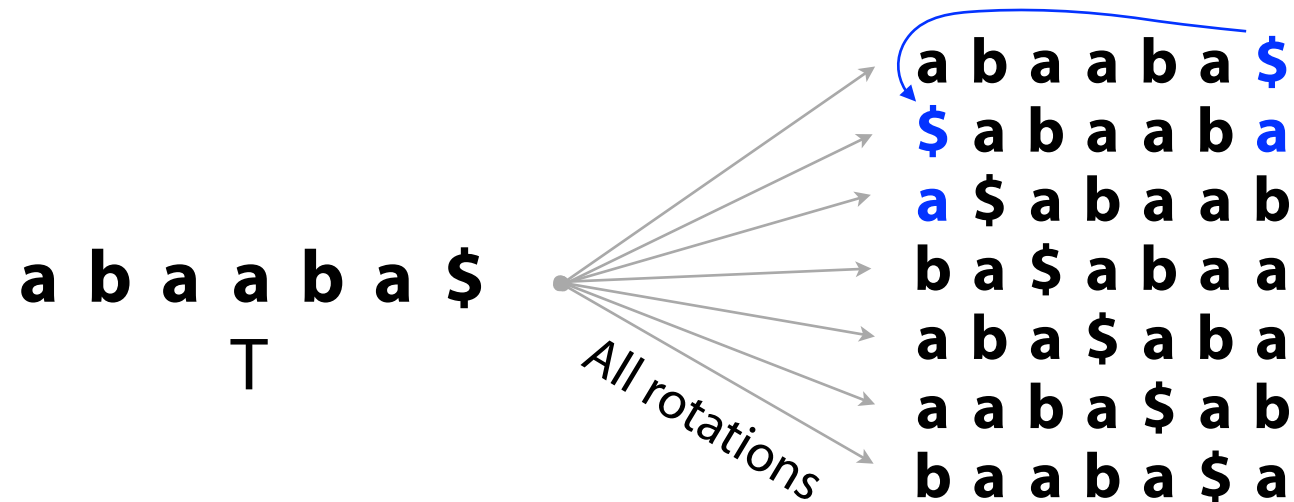
Burrows-Wheeler Transform

1) Build all **text rotations** of the input string



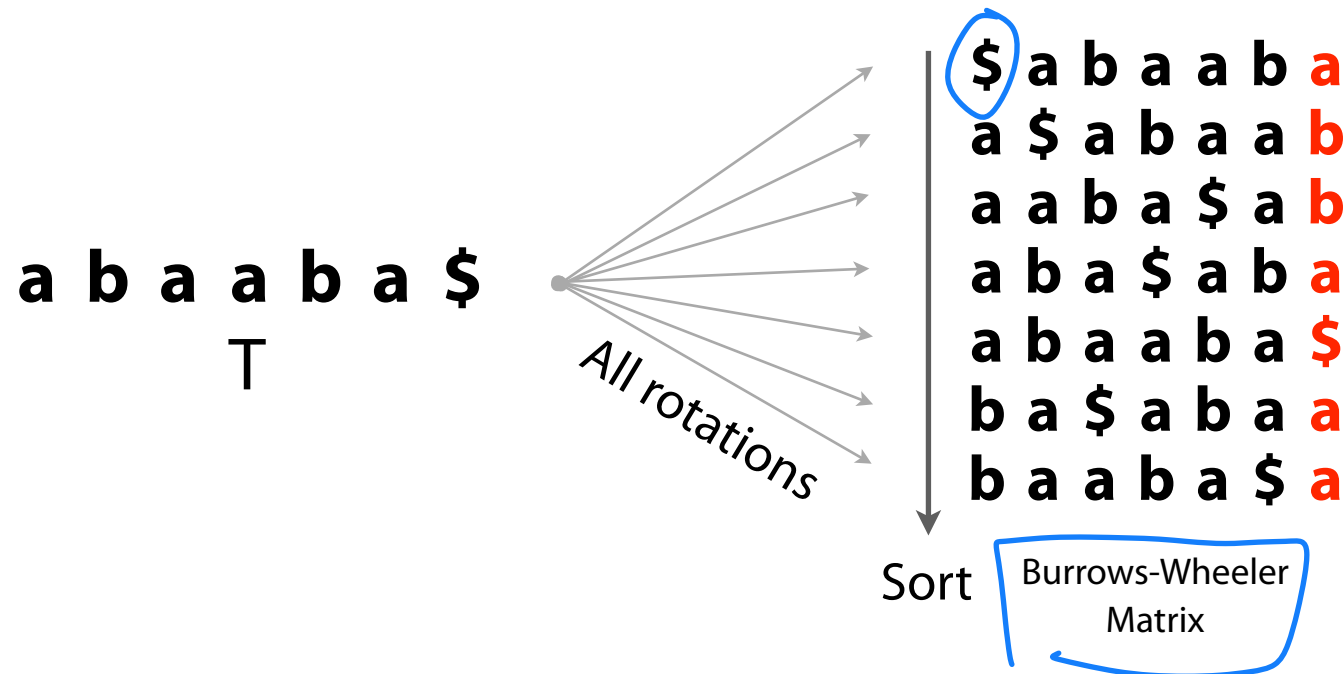
Burrows-Wheeler Transform

1) Build all **text rotations** of the input string



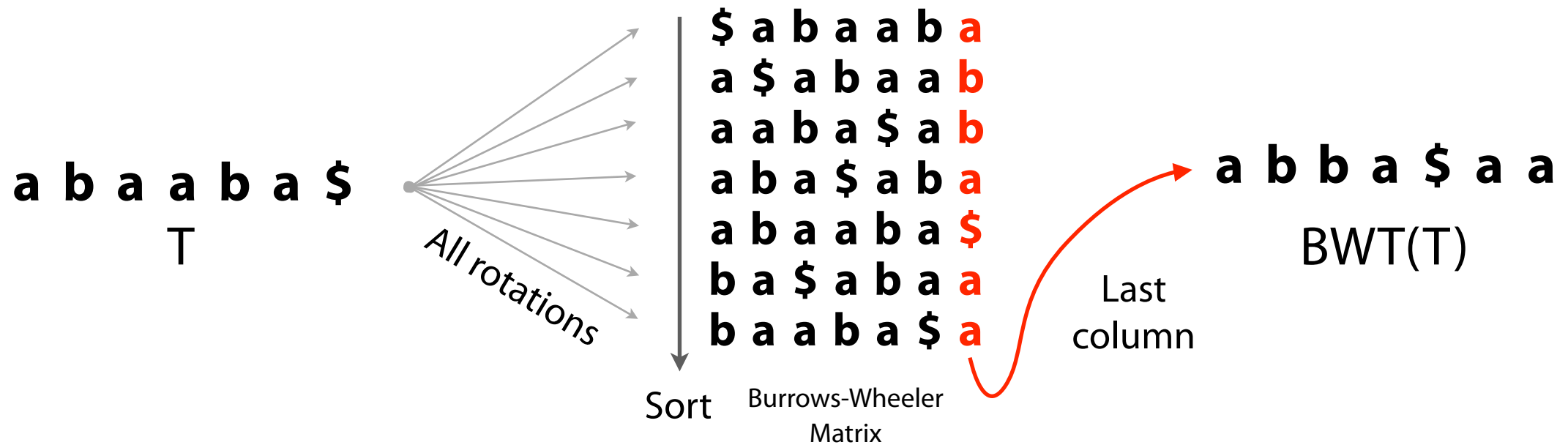
Burrows-Wheeler Transform

2) Sort all **text rotations** of the input string lexicographically



Burrows-Wheeler Transform

3) Take the last column. This is our **Burrows-Wheeler Transform**



Burrows-Wheeler Transform

- (1) Build all rotations
- (2) Sort all rotations
- (3) Take last column

T = c a r \$

\$ (a r

r \$ c a

a r \$ c



\$ (a r

a r \$ c

c a r \$

r \$ c a

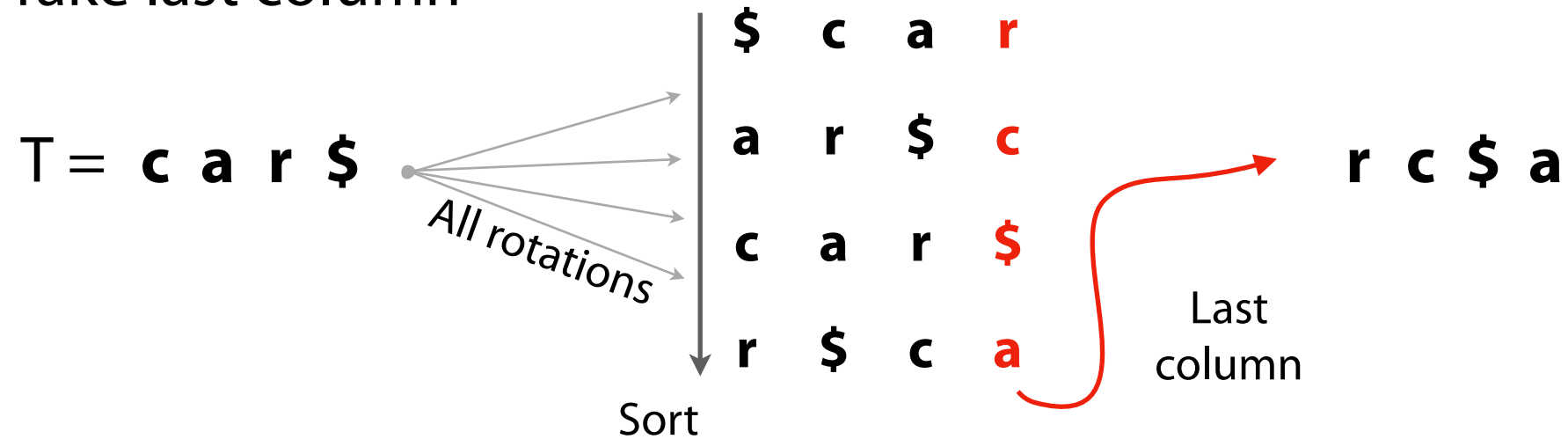


r c \$ a



Burrows-Wheeler Transform

- (1) Build all rotations
- (2) Sort all rotations
- (3) Take last column



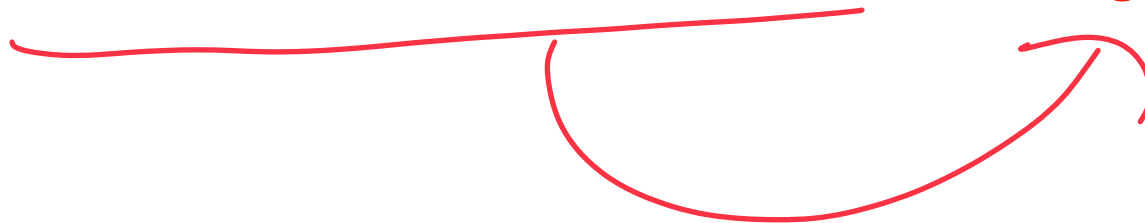
Assignment 8: a_bwt

Learning Objective:

Implement the Burrows-Wheeler Transform on text

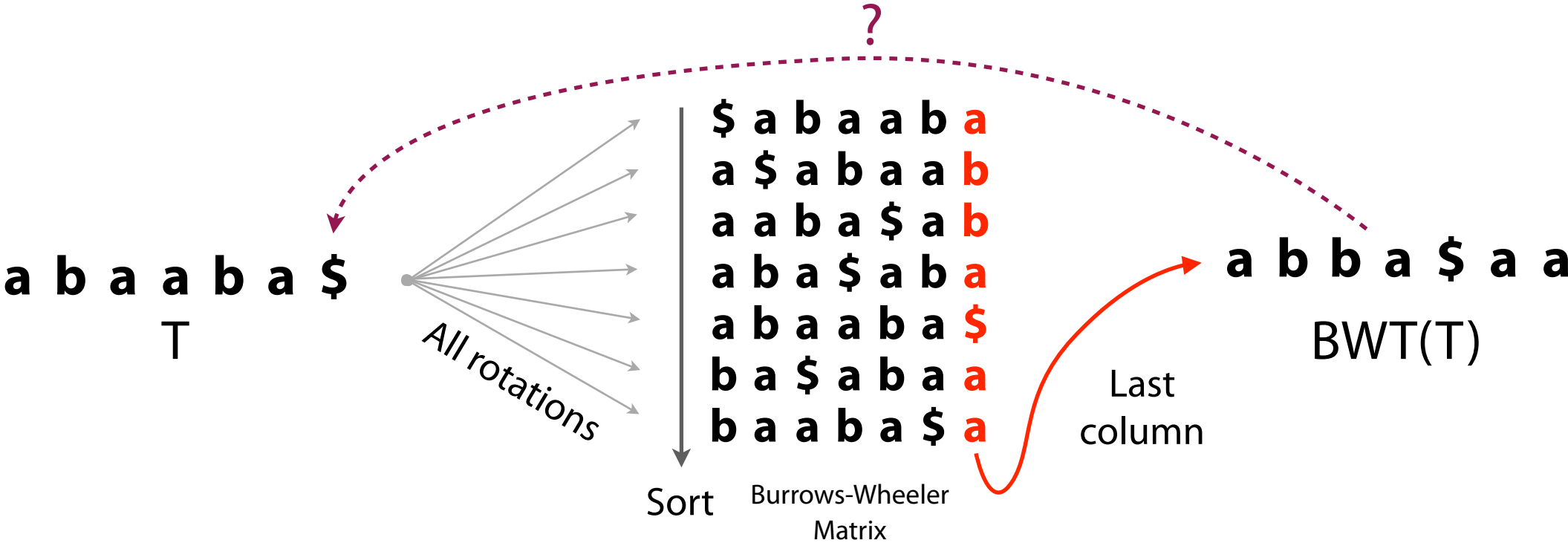
Reverse the Burrows-Wheeler Transform to reproduce text

Consider: How can the BWT be stored *smaller* than the original text?



Burrows-Wheeler Transform

How to reverse the BWT?



Burrows-Wheeler Transform

BWT(T) = **r c \$ a**

T = **c a r \$**

1) Prepend the BWT

r
c
\$
a

r \$
c a
\$ r
a c

2) Sort

r \$ c
c a r
\$ c a
a r \$

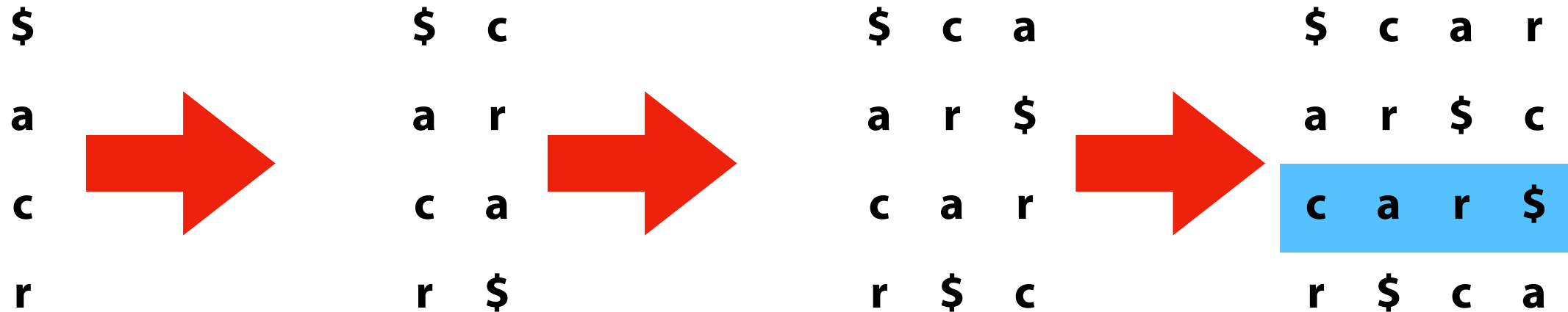
3) Repeat!

r \$ c a
c a r \$
\$ c a r
a r \$ c

Burrows-Wheeler Transform

BWT(T) = **r c \$ a** T = **c a r \$**

- 1) Prepend the BWT as a column
- 2) Sort the full matrix as rows
- 3) Repeat 1 and 2 until full matrix
- 4) Pick the row ending in '\$'

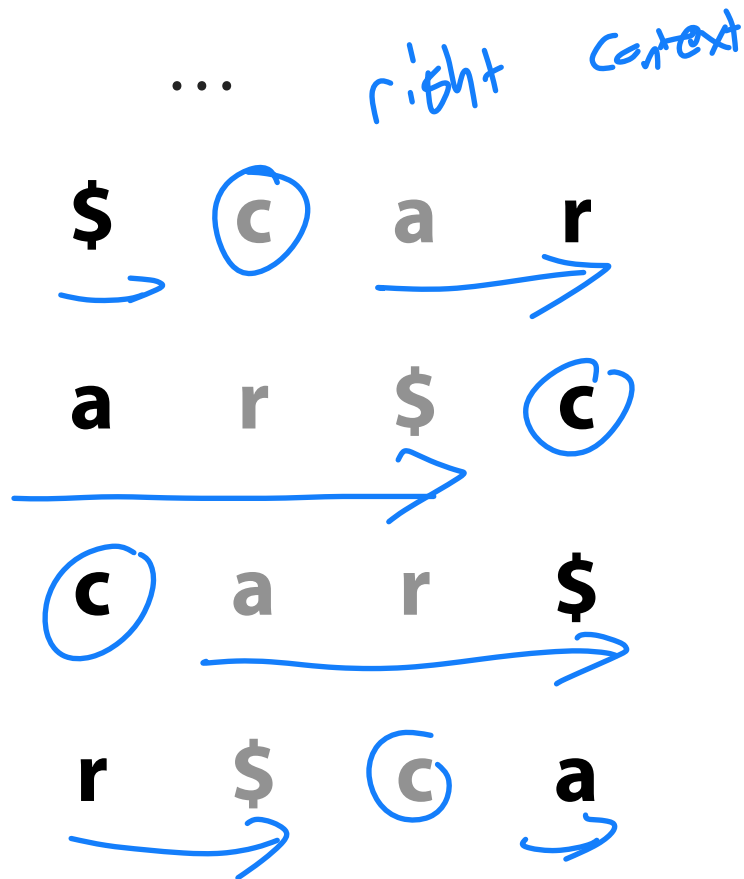


Burrows-Wheeler Transform

This works because we are storing sorted rotations

Just before '\$', there was an 'r'.

Just before 'a', there was an 'c'.



$$\text{BWT}(T) = \underline{\text{r c \$ a}}$$

$$T = \text{c a r \$}$$

Sorted row

r \$

c a

\$ c

r a

Burrows-Wheeler Transform

This works because we are storing **sorted rotations**

Just before '\$c', there was an 'r'.

Just before 'ar', there was an 'c'.

...

\$ c a r

a r \$ c

c a r \$

r \$ c a

BWT(T) = r c \$ a

T = c a r \$

\$ c

a r

c a

r \$

Burrows-Wheeler Transform

The **right context** is the wrap-around text

'r' has right context '\$ca'.

'c' has right context 'ar\$'.

...

\$ c a r

a r \$ c

c a r \$

r \$ c a

BWT(T) = r c \$ a

T = c a r \$



⤵ \$ c a

⤵ a r \$

\$ c a r

4 r \$ c

Burrows-Wheeler Transform

What is the right context of **a p p l e \$** ?



le\$97

Burrows-Wheeler Transform

What is the right context of **a p**p**l e \$** ?

l e \$ a p

A letter always has the same right context.

\$	a	p	p	l	e
a	p	p	l	e	\$
e	\$	a	p	p	l
l	e	\$	a	p	p
p	l	e	\$	a	p
p	p	l	e	\$	a

Burrows-Wheeler Transform: T-ranking

To continue, we have to be able to uniquely identify each character in our text.

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

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

Ranks aren't explicitly stored; they are just for illustration

Burrows-Wheeler Transform

BWM with T-ranking:

\$	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

Burrows-Wheeler Transform

BWM with T-ranking:

	F						L
3	\$	a ₀	b ₀	a ₁	a ₂	b ₁	a ₃
1	a ₃	\$	a ₀	b ₀	a ₁	a ₂	b ₁
2	a ₁	a ₂	b ₁	a ₃	\$	a ₀	b ₀
5	a ₂	b ₁	a ₃	\$	a ₀	b ₀	a ₁
6	a ₀	b ₀	a ₁	a ₂	b ₁	a ₃	\$
4	b ₁	a ₃	\$	a ₀	b ₀	a ₁	a ₂
0	b ₀	a ₁	a ₂	b ₁	a ₃	\$	a ₀

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

The order of characters in *F* & *L* are the same

Burrows-Wheeler Transform

BWM with T-ranking:

<i>F</i>							<i>L</i>
\$	a ₀	b ₀	a ₁	a ₂	b ₁	a₃	
a₃	\$	a ₀	b ₀	a ₁	a ₂	b ₁	
a₁	a ₂	b ₁	a ₃	\$	a ₀	b ₀	
a₂	b ₁	a ₃	\$	a ₀	b ₀	a₁	
a₀	b ₀	a ₁	a ₂	b ₁	a ₃	\$	
b ₁	a ₃	\$	a ₀	b ₀	a ₁	a₂	
b ₀	a ₁	a ₂	b ₁	a ₃	\$	a₀	

Look at first and last columns, called *F* and *L* (and look at just the **as**)

Burrows-Wheeler Transform

BWM with T-ranking:

<i>F</i>							<i>L</i>
\$	a ₀	b ₀	a ₁	a ₂	b ₁	a ₃	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	b ₀
a ₂	b ₁	a ₃	\$	a ₀	b ₀	a ₁	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	a ₂
b ₀	a ₁	a ₂	b ₁	a ₃	\$	a ₀	a ₀

Look at first and last columns, called *F* and *L* (and look at just the **as**)

as occur in the same order in *F* and *L*. As we look down columns, in both cases we see: **a₃, a₁, a₂, a₀**

Burrows-Wheeler Transform

BWM with T-ranking:

<i>F</i>						<i>L</i>
\$	a ₀	b ₀	a ₁	a ₂	b ₁	a ₃
a ₃	\$	a ₀	b ₀	a ₁	a ₂	b₁
a ₁	a ₂	b ₁	a ₃	\$	a ₀	b₀
a ₂	b ₁	a ₃	\$	a ₀	b ₀	a ₁
a ₀	b ₀	a ₁	a ₂	b ₁	a ₃	\$
b₁	a ₃	\$	a ₀	b ₀	a ₁	a ₂
b₀	a ₁	a ₂	b ₁	a ₃	\$	a ₀

Same with **bs**: **b₁**, **b₀**

Burrows-Wheeler Transform: LF Mapping

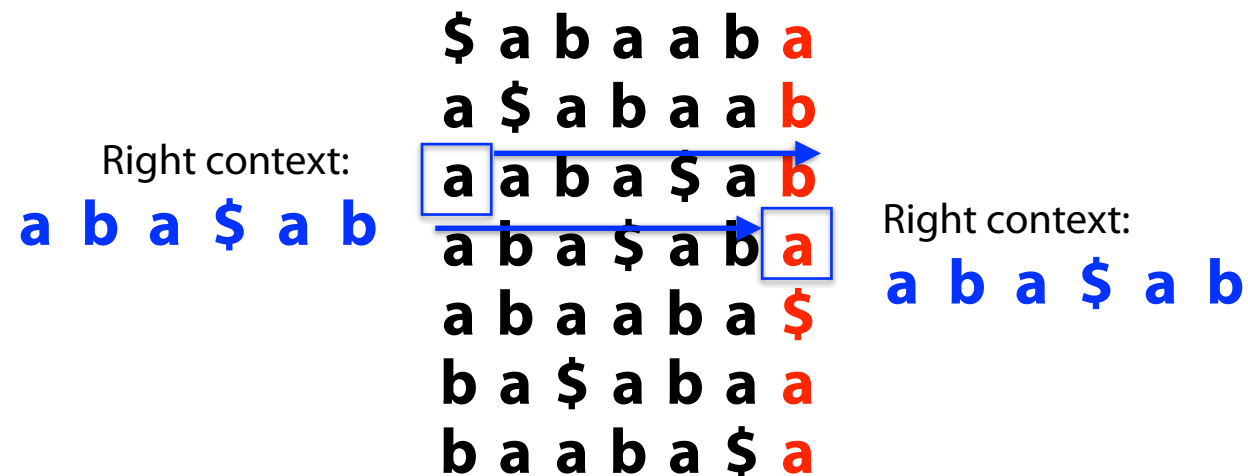
BWM with T-ranking:

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 (i.e. have same rank)

Burrows-Wheeler Transform: LF Mapping

Why does this work?



These characters have the same right contexts!

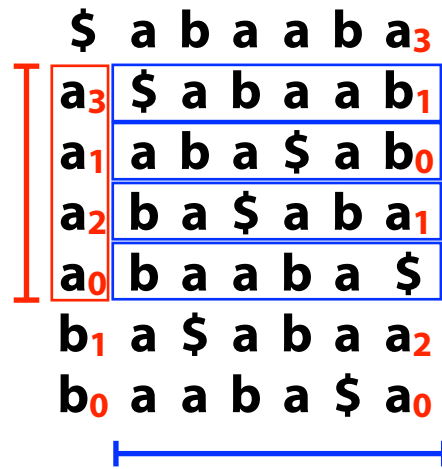
These characters *are the same character!* **a₀ b₀ a₁ a₂ b₁ a₃ \$**



Burrows-Wheeler Transform: LF Mapping

Why does this work?

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



They're sorted by right-context



They're sorted by right-context

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

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

Any ranking we give to characters in T will match in F and L

Burrows-Wheeler Transform: LF Mapping

LF Mapping can be used to recover our original text too!

Given BWT = **a₃ b₁ b₀ a₁ \$ a₂ a₀**

What is L? BWT is L

What is F? Use content of BWT:

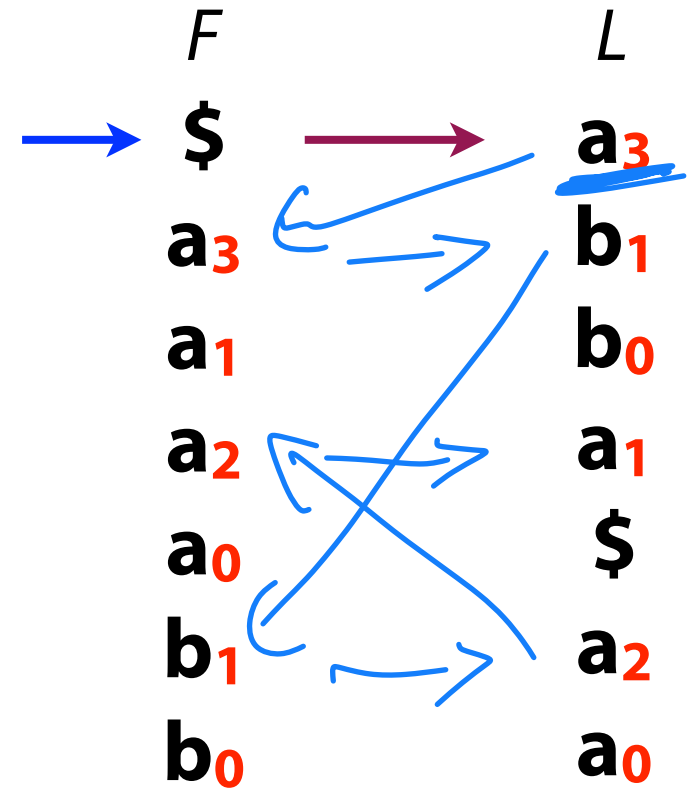
\$	a	a	a	a	b	b	7	\$
	₃	₁	₂	₀	₁	₀	4	9
							2	5

Burrows-Wheeler Transform: LF Mapping

LF Mapping can be used to recover our original text too!

Start in first row. F must have \$.

L contains character just prior to \$: a_3



$a_1, a_2, b_1, a_3, \$$

Burrows-Wheeler Transform: LF Mapping

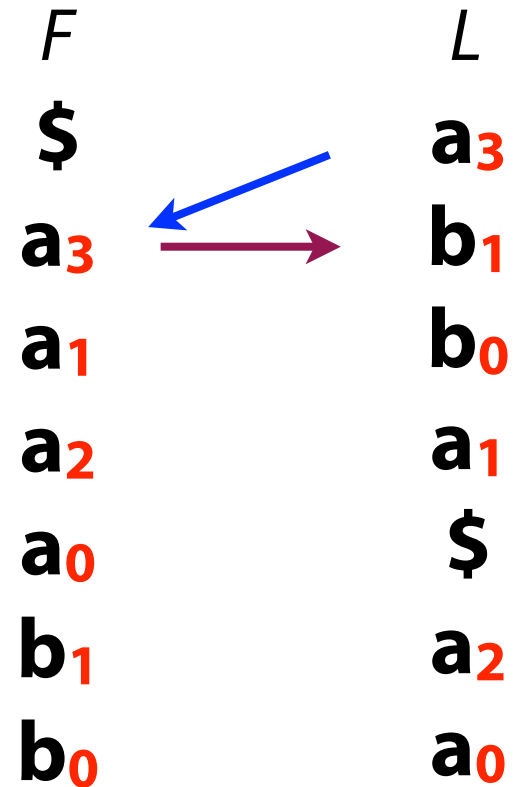
LF Mapping can be used to recover our original text too!

Start in first row. F must have \$.

L contains character just **prior** to \$: a_3

Jump to row *beginning* with a_3 .

L contains character just **prior** to a_3 : b_1 .



Burrows-Wheeler Transform: LF Mapping

LF Mapping can be used to recover our original text too!

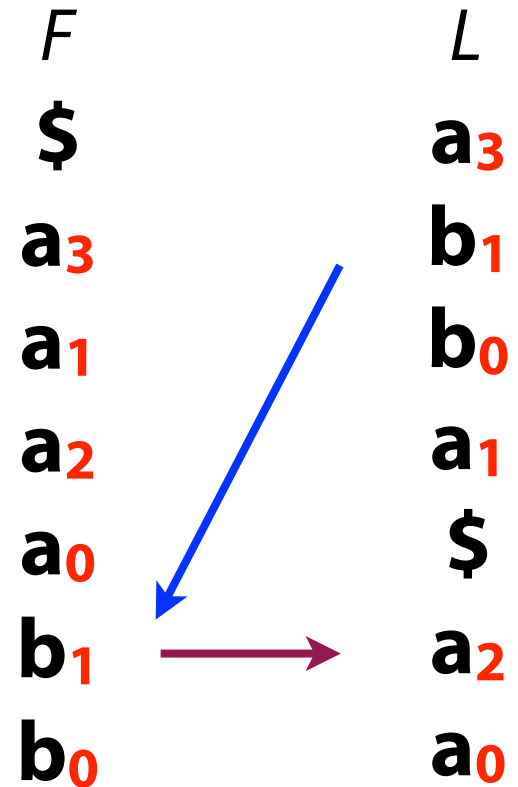
Start in first row. F must have \$.

L contains character just **prior** to \$: a_3

Jump to row *beginning* with a_3 .

L contains character just **prior** to a_3 : b_1 .

Repeat for b_1 , get a_2



Burrows-Wheeler Transform: LF Mapping

LF Mapping can be used to recover our original text too!

Start in first row. F must have \$.

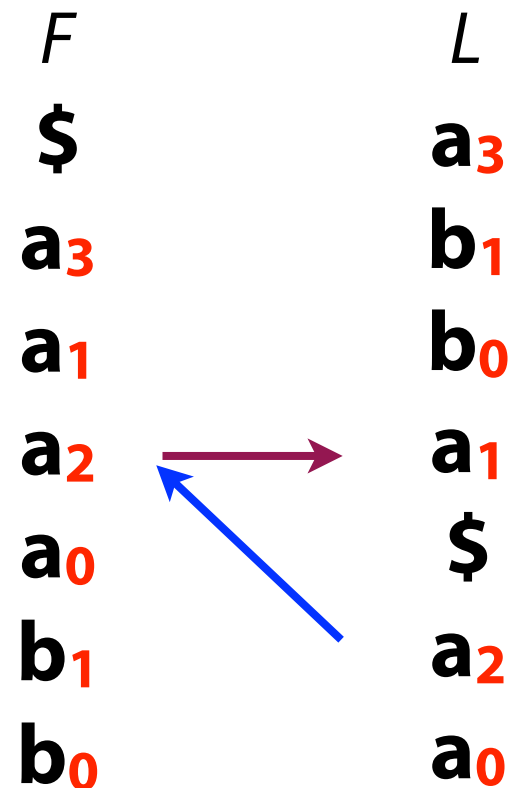
L contains character just **prior** to \$: a_3

Jump to row *beginning* with a_3 .

L contains character just **prior** to a_3 : b_1 .

Repeat for b_1 , get a_2

Repeat for a_2 , get a_1



Burrows-Wheeler Transform: LF Mapping

LF Mapping can be used to recover our original text too!

Start in first row. F must have \$.

L contains character just **prior** to \$: a_3

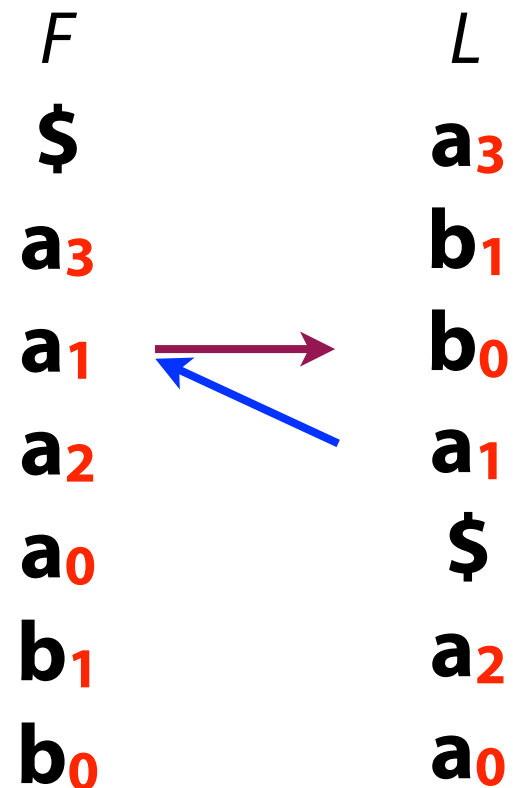
Jump to row *beginning* with a_3 .

L contains character just **prior** to a_3 : b_1 .

Repeat for b_1 , get a_2

Repeat for a_2 , get a_1

Repeat for a_1 , get b_0



Burrows-Wheeler Transform: LF Mapping

LF Mapping can be used to recover our original text too!

Start in first row. F must have \$.

L contains character just **prior** to \$: a_3

Jump to row *beginning* with a_3 .

L contains character just **prior** to a_3 : b_1 .

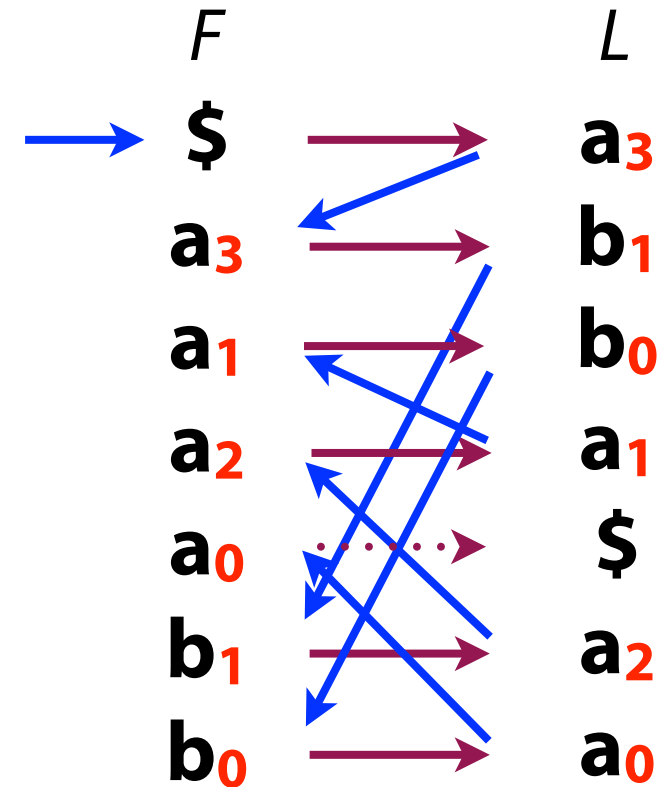
Repeat for b_1 , get a_2

Repeat for a_2 , get a_1

Repeat for a_1 , get b_0

Repeat for b_0 , get a_0

Repeat for a_0 , get \$ (done)

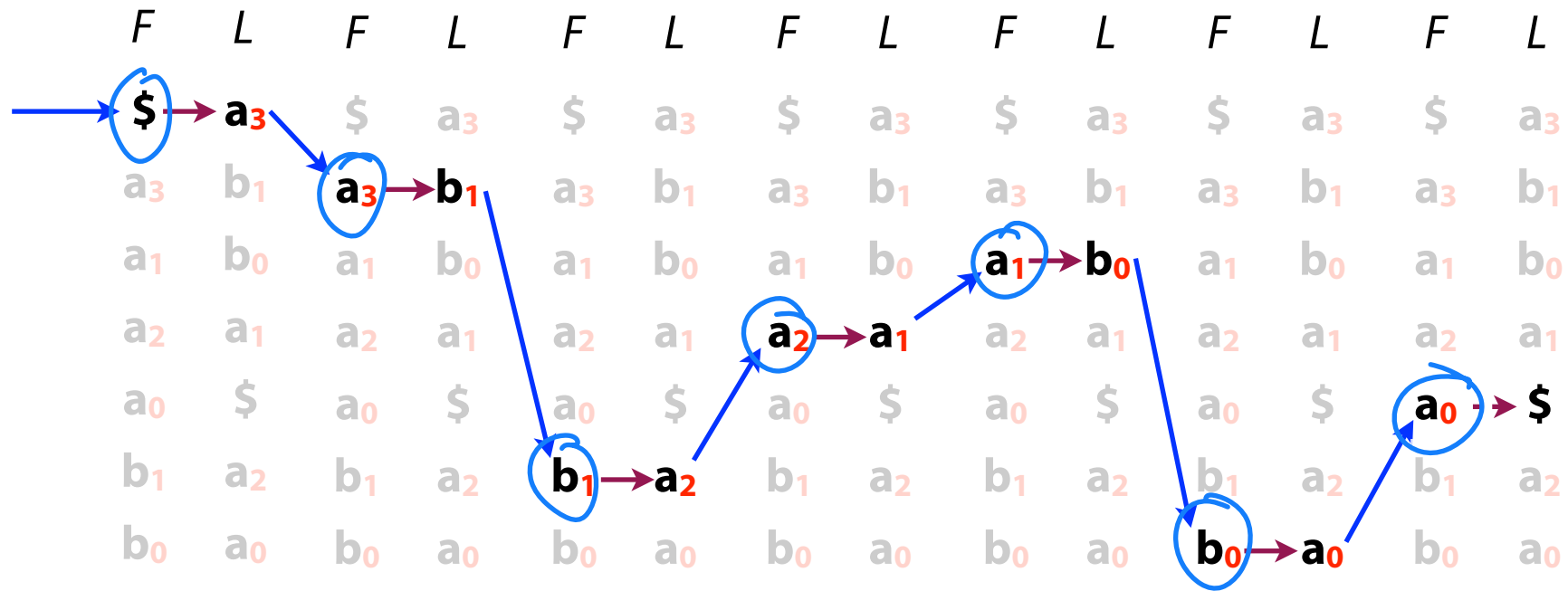


In reverse order, $T = \underline{a_0 b_0 a_1 a_2 b_1 a_3 \$}$

Burrows-Wheeler Transform: LF Mapping



Another way to visualize:



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

Assignment 8: a_bwt

Learning Objective:

Implement the Burrows-Wheeler Transform on text

Reverse the Burrows-Wheeler Transform to reproduce text

Consider: You can use either LF mapping or prepend-sort to reverse. Which do you think would be easier to implement (or more efficient)?

Burrows-Wheeler Transform: A better ranking

Any ranking we give to characters in T will match in F and L

T-Rank: Order by T

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

F-Rank: Order by F

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

What is good about F-rank?

Burrows-Wheeler Transform: A better ranking

T = a b b c c d \$

What is the BWM index for my first instance of C? (C₀) [0-base for answer]

Need

characters!

2 \$
1 A_s
2 B_s
2 C_s

0 \$
1 a
2 b
3 b
4 → c
5 d

add all prev characters

→ L odd → index L[4]

Burrows-Wheeler Transform: A better ranking

T = **a b b c c d \$**

What is the BWM index for my first instance of C? (**C₀**) [0-base for answer]

	<i>F</i>						<i>L</i>
0	\$	a	b	b	c	c	d
1	a	b	b	c	c	d	\$
2	b	b	c	c	d	\$	a
3	b	c	c	d	\$	a	b
4	c	c	d	\$	a	b	b
	c	d	\$	a	b	b	c
	d	\$	a	b	b	c	c

Burrows-Wheeler Transform: A better ranking

T = **a b b c c d \$**

What is the BWM index for my first instance of C? (**C₀**) [0-base for answer]

Skip '\$' (1)

Skip 'A' (1)

Skip 'B' (2)

Look-up F[**4**] / L[**4**]

<i>F</i>							<i>L</i>
\$	a	b	b	c	c		d
a	b	b	c	c	d		\$
b	b	c	c	d	\$		a
b	c	c	d	\$	a		b
c	c	d	\$	a	b		b
c	d	\$	a	b	b		c
d	\$	a	b	b	c		c

Burrows-Wheeler Transform: A better ranking

Say T has 300 As, 400 Cs, 250 Gs and 700 Ts and $\$ < A < C < G < T$

What is the BWM index for my 100th instance of G? (G₉₉) [0-base for answer]

1 \$

300 As

400 Cs

99 G

200

~~250~~
~~330~~
749
800



Burrows-Wheeler Transform: A better ranking

Say T has 300 **A**s, 400 **C**s, 250 **G**s and 700 **T**s and $\$ < \mathbf{A} < \mathbf{C} < \mathbf{G} < \mathbf{T}$

What is the BWM index for my 100th instance of **G**? (**G**₉₉) [0-base for answer]

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

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

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

Skip first 99 rows starting with **G** (99 rows)

Skip all before me

Answer: skip 800 rows -> **index 800 contains my 100th G**

With a little preprocessing we can find any character in $O(1)$ time!



Join Code: 225

FM Index

(Next week's material)

An index combining the BWT with a few small auxiliary data structures

Core of index is **first (F)** and **last (L) rows** from BWM:

L is the same size as **T**

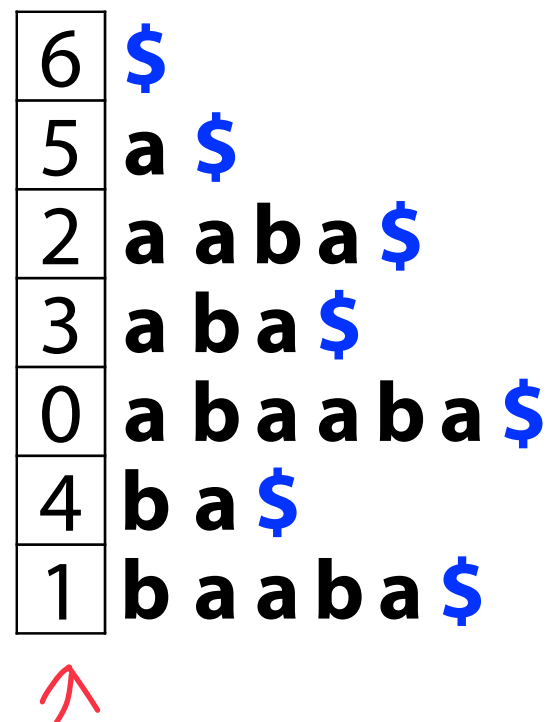
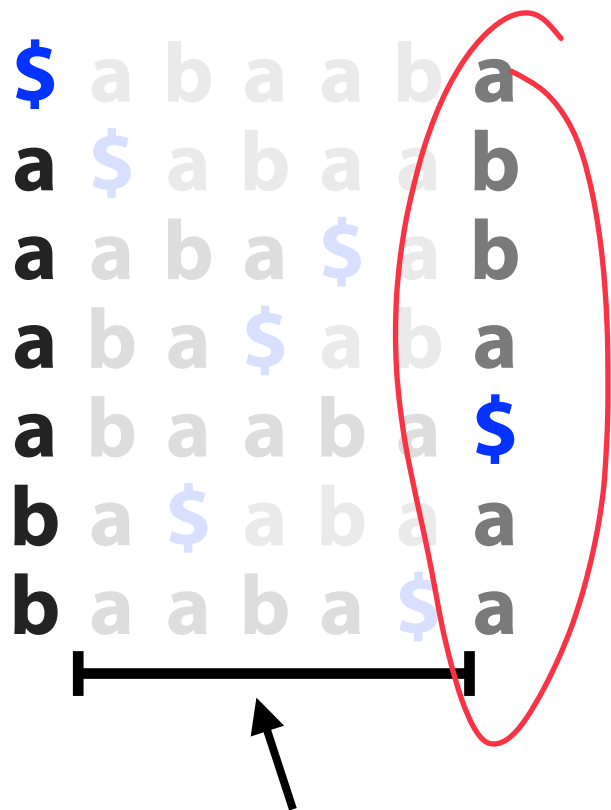
~~F~~ can be represented as array of $|\Sigma|$ integers (or not stored at all!)

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

We're discarding ~~T~~ — we can recover it from L!

FM Index: Querying

The BWM is a lot like the suffix array — maybe we can query the same way?



We don't have these columns, and we don't have T.

FM Index: Querying

4 as

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

Start with shortest suffix, then match successively longer suffixes

Some where here are my AS

$O(1)$

1 \$
2
3
4

$P = \mathbf{aba}$

F						L
\$	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_1
a_3	b	a	a	b	a	\$
b	a	\$	a	b	a	a_2
b	a	a	b	a	\$	a_3

Start w/ last character

1) Get all as in F

FM Index: Querying

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

Start with shortest suffix, then match successively longer suffixes

$P = \text{aba}$

Easy to find all the rows
beginning with **a**

	F					L	
	\$	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_1
	a_3	b	a	a	b	a	\$
	b	a	\$	a	b	a	a_2
	b	a	a	b	a	\$	a_3

Some where in here
could be "b"

FM Index: Querying

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

$P = \mathbf{aba}$

<i>F</i>						<i>L</i>
\$	a	b	a	a	b	a₀
a₀	\$	a	b	a	a	b₀
a₁	a	b	a	\$	a	b₁
a₂	b	a	\$	a	b	a₁
a₃	b	a	a	b	a	\$
b₀	a	\$	a	b	a	a₂
b₁	a	a	b	a	\$	a₃

← Look at those rows in *L*.

FM Index: Querying

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

$P = \mathbf{aba}$

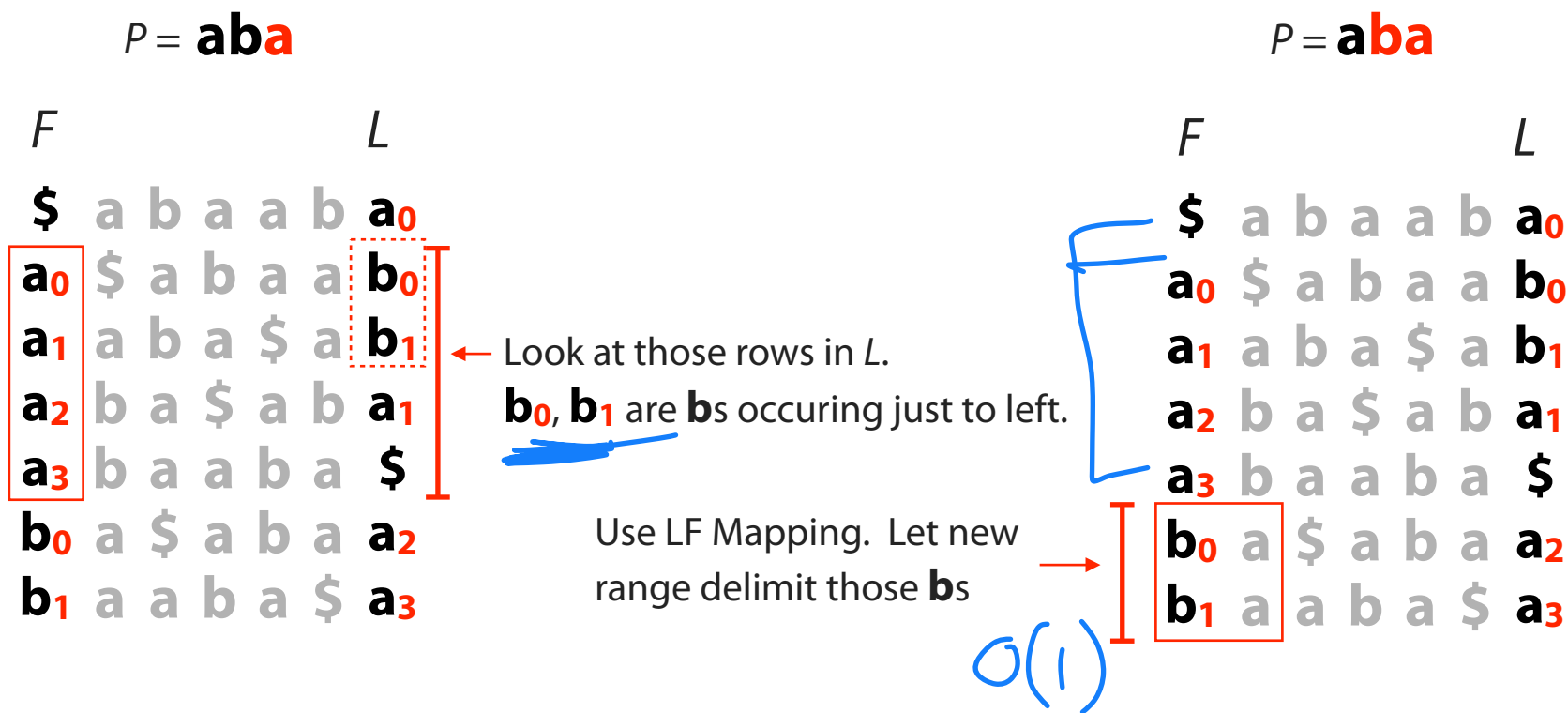
<i>F</i>							<i>L</i>
\$	a	b	a	a	b		a ₀
a ₀	\$	a	b	a	a		b ₀
a ₁	a	b	a	\$	a		b ₁
a ₂	b	a	\$	a	b		a ₁
a ₃	b	a	a	b	a		\$
b ₀	a	\$	a	b	a		a ₂
b ₁	a	a	b	a	\$		a ₃

← Look at those rows in *L*.

b₀, **b**₁ are **b**s occuring just to left.

FM Index: Querying

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



Note: We still aren't storing the characters in grey, we just know they exist.

FM Index: Querying

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

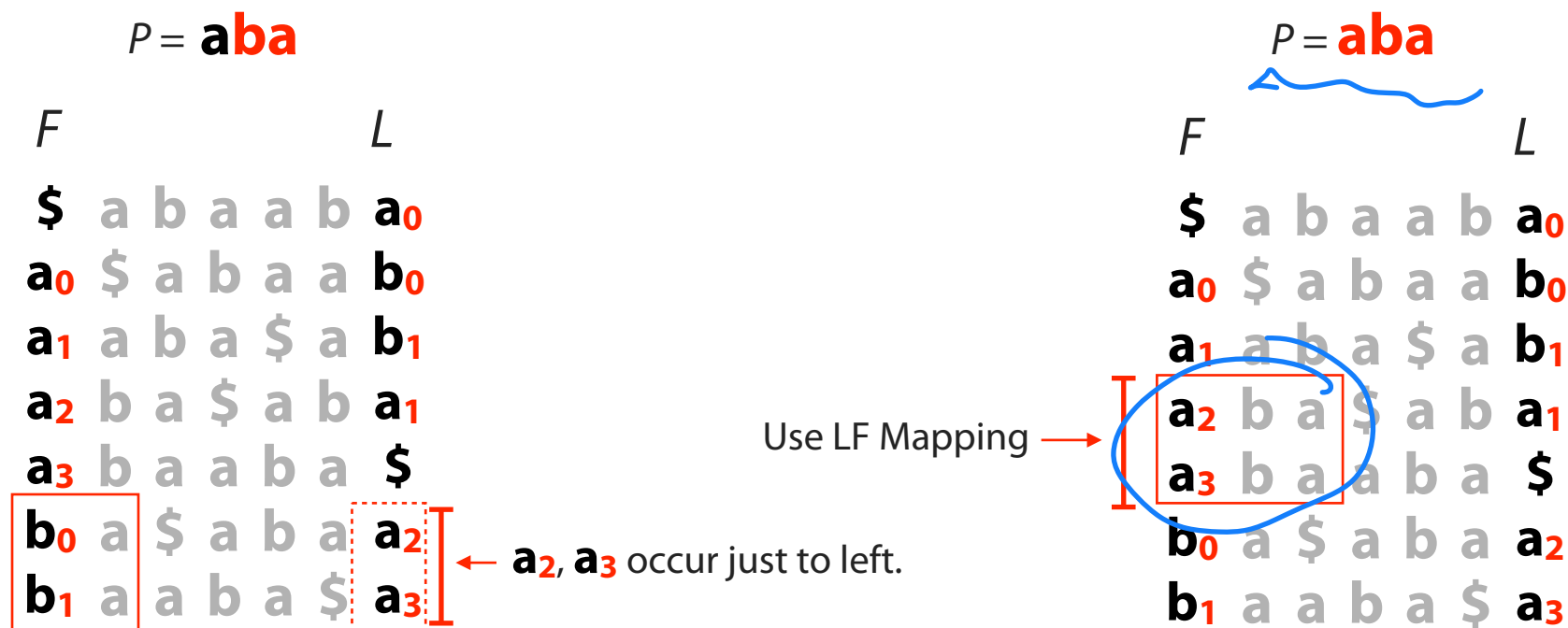
$P = \mathbf{aba}$

<i>F</i>						<i>L</i>
\$	a	b	a	a	b	a₀
a₀	\$	a	b	a	a	b₀
a₁	a	b	a	\$	a	b₁
a₂	b	a	\$	a	b	a₁
a₃	b	a	a	b	a	\$
b₀	a	\$	a	b	a	a₂
b₁	a	a	b	a	\$	a₃

← **a₂**, **a₃** occur just to left.

FM Index: Querying

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



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

FM Index: Querying

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

$P = \mathbf{bba}$

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 \$

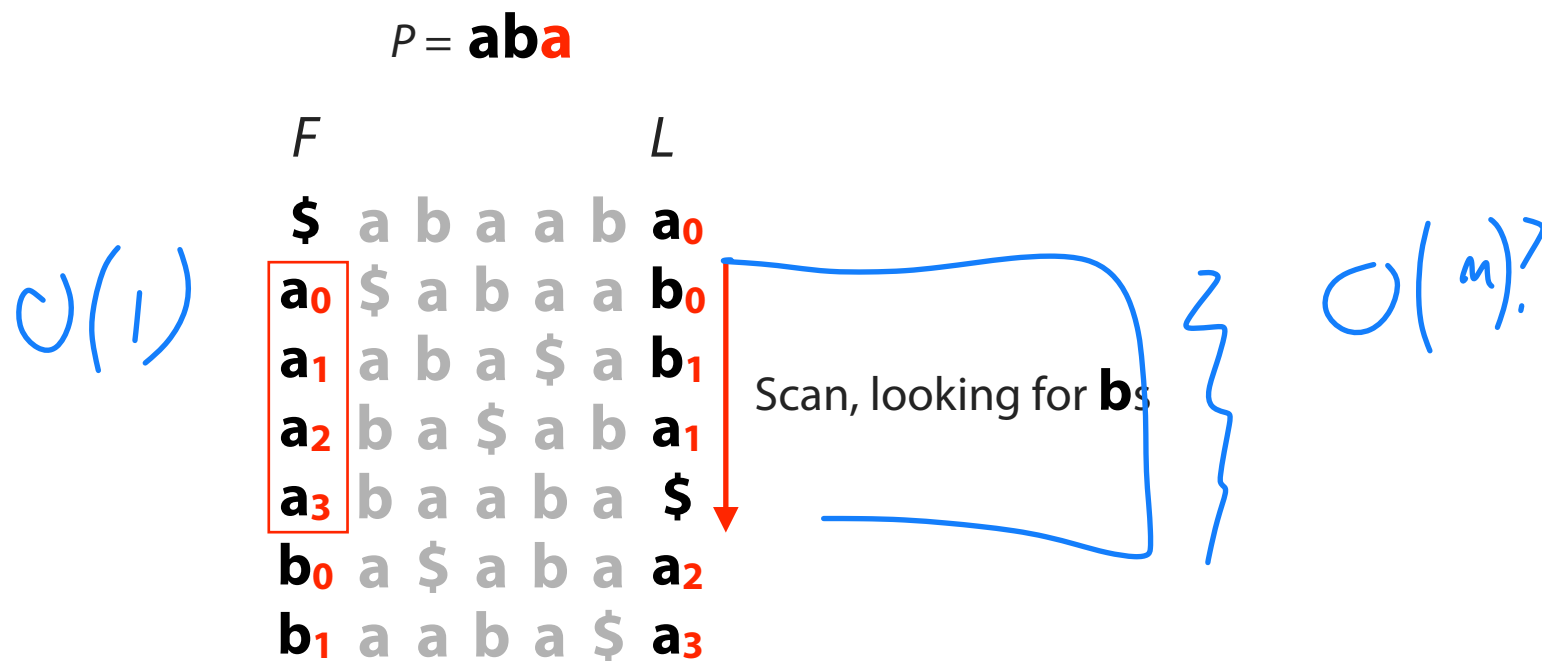
Rows with **ba** prefix

$\mathbf{b_0}$	a	\$	a	b	a	$\mathbf{a_2}$
$\mathbf{b_1}$	a	a	b	a	\$	$\mathbf{a_3}$

← No **bs**!

FM Index: Querying

Problem 1: If we *scan* characters in the last column, that can be slow, $O(m)$



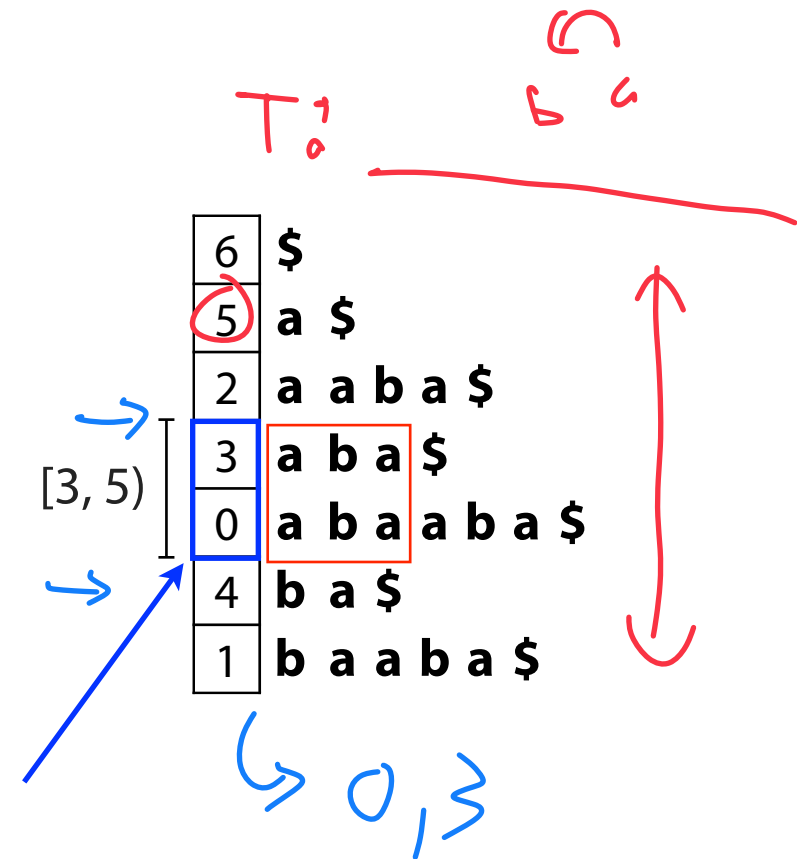
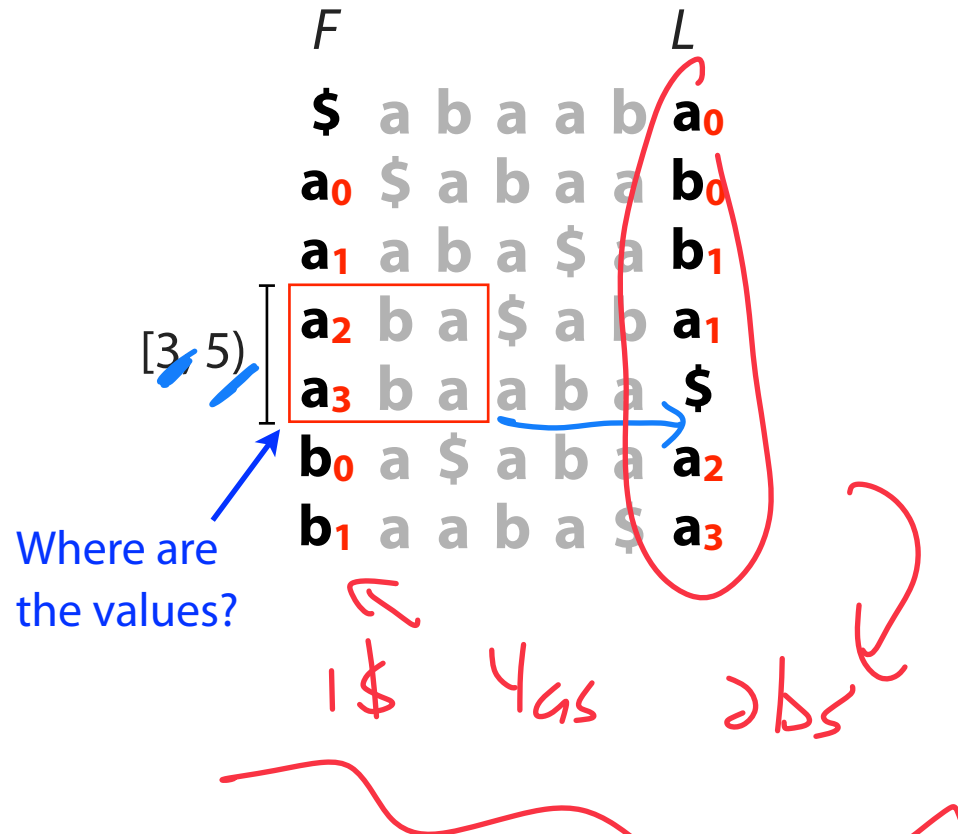


FM Index: Querying

Problem 2: We don't immediately know *where* the matches are in T...

$P = \mathbf{aba}$

Got the same range, $[3, 5)$, we would have got from suffix array

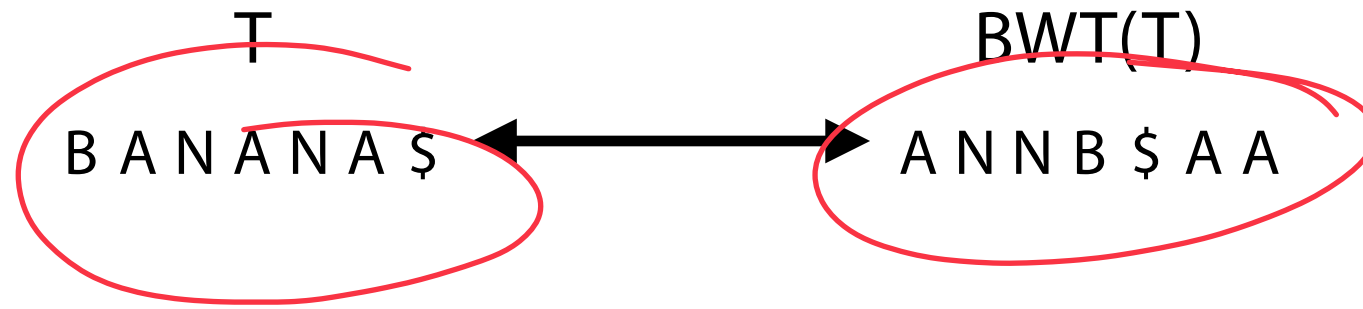




Bonus Slides

Burrows-Wheeler Transform

Reversible permutation of the characters of a string



1) How to encode?

2) How to decode?

3) How is it useful for compression?

4) How is it useful for search?

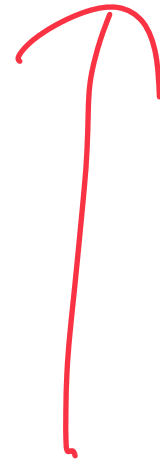
Burrows-Wheeler Transform

Tomorrow_and_tomorrow_and_tomorrow

w\$wdd__nnooaattTmmrrrrrrrooo__ooo

It_was_the_best_of_times_it_was_the_worst_of_times\$

s\$esttssffteww_hhmmbootttt_ii__woeearessIi_____



“bzip”: compression w/ a BWT to better organize text

Burrows-Wheeler Transform

orrow_and_tomorrow_and_tomorrow\$tom
ow\$tomorrow_and_tomorrow_and_tomor^r
ow_and_tomorrow\$tomorrow_and_tomor^r
ow_and_tomorrow_and_tomorrow\$tomor^r
row\$tomorrow_and_tomorrow_and_tomor^r
row_and_tomorrow\$tomorrow_and_tomor^r
row_and_tomorrow_and_tomorrow\$tomor^r
rrow\$tomorrow_and_tomorrow_and_tomo

Ordered by the context to the right of each character

Burrows-Wheeler Transform

In English (and most languages), the next character in a word is not independent of the previous.

In general, if text structured BWT(T) 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-Wheeler Transform

Lets compare the SA with the BWT...

T = a b a a b a \$

6
5
2
3
0
4
1

SA(T)

Suffix Array is $O(m)$

\$	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)

Burrows-Wheeler Transform

Lets compare the SA with the BWT...

T = a b a a b a \$

6
5
2
3
0
4
1

SA(T)

Suffix Array is $O(m)$



a b b a \$ a a
1 9
2 8
1 6
1 5
0 9

BWT(T)

BWT is $O(m)$

The BWT has a better constant factor!

Burrows-Wheeler Transform

BWM is related 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)

Same order 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 \$

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 \$

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



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”

