String Algorithms and Data Structures
Burrows-Wheeler Transform

CS 199-225

October 28, 2024

Brad Solomon

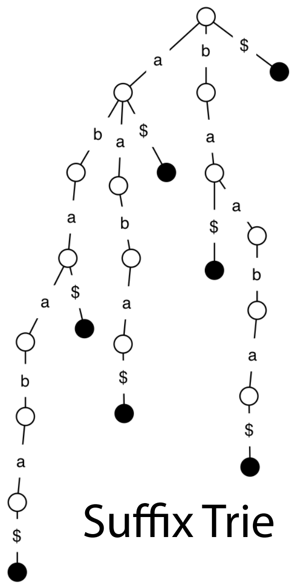
UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

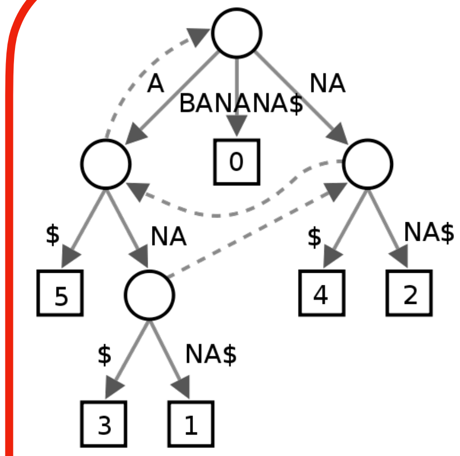
Exact pattern matching *w/ indexing*

There are many data structures built on ***suffixes***

We have now seen both of these data structures



Suffix Trie



Suffix Tree

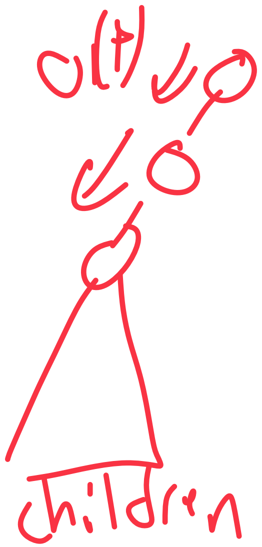
6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Suffix Array

$ B A N A N A
A $ B A N A N
A N A $ B A N
A N A N A $ B
B A N A N A $
N A $ B A N A
N A N A $ B A

FM Index

Exact pattern matching *w/ indexing*



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

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

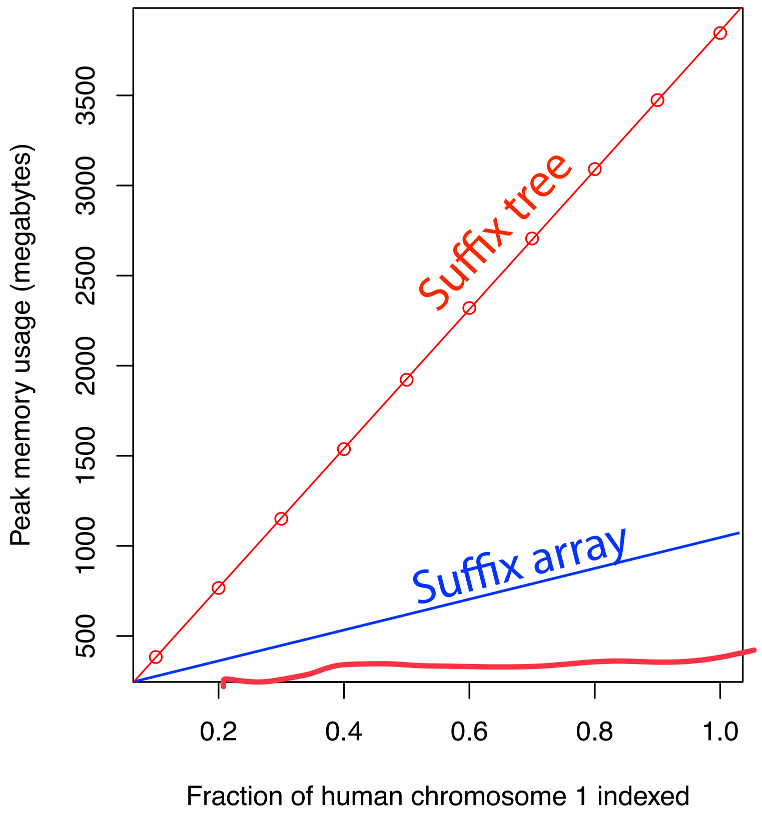
Exact pattern matching *w/ indexing*

	Suffix tree	Suffix array	Suffix array (Not covered)
Time: Does P occur?	$O(n)$	$O(n \log m)$	$O(n + \log m)$
Time: Report k locations of P	$O(n + k)$	$O(n \log m + k)$	$O(n + \log m)$
Space	$O(m)$	$O(m)$	

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

Suffix tree vs suffix array: size

The suffix array has a smaller constant factor than the tree



Suffix tree: ~16 bytes per character

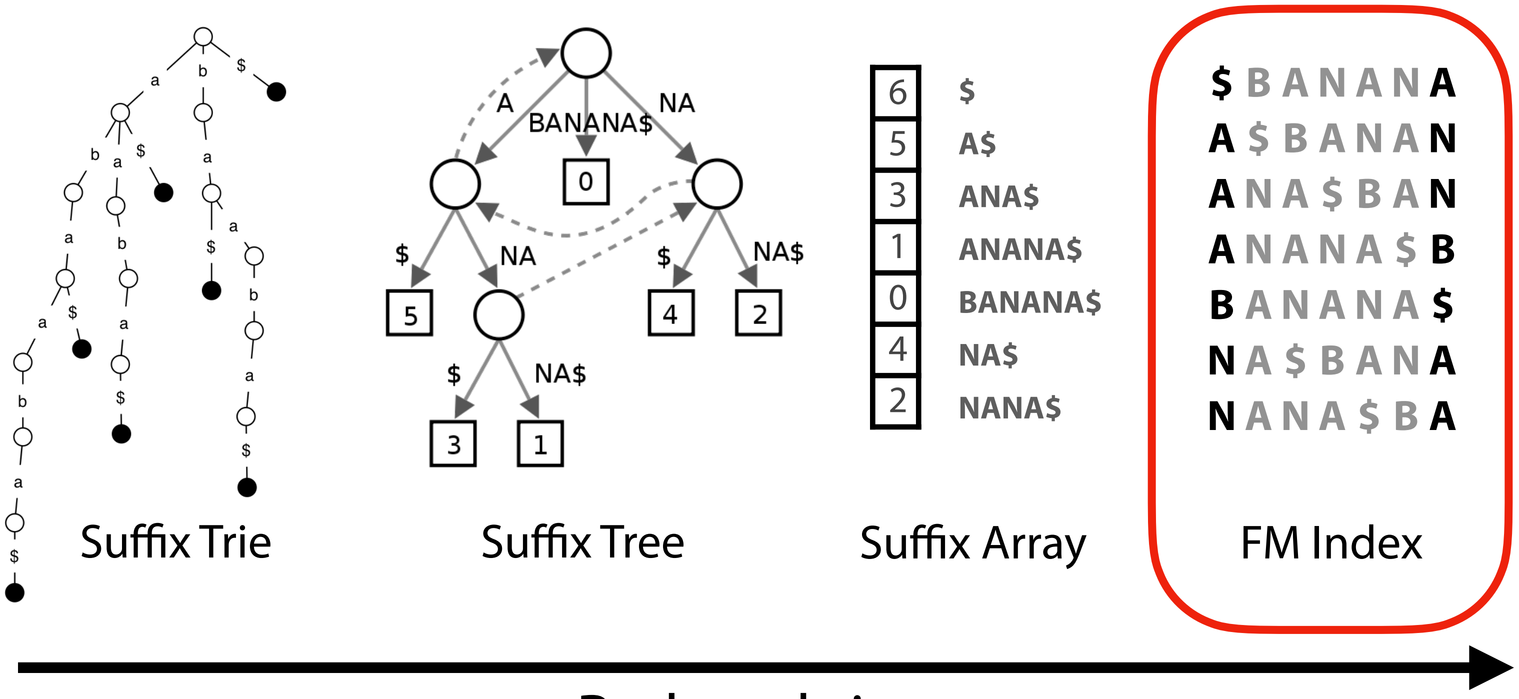
Suffix array: ~4 bytes per character

Raw text: 2 bits per character

Exact pattern matching *w/ indexing*

There are many data structures built on ***suffixes***

The FM index is a compressed self-index (smaller* than original text)!



Reduced size

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		BWT(T)
B A N A N A $	⟷	A N N B $ A A

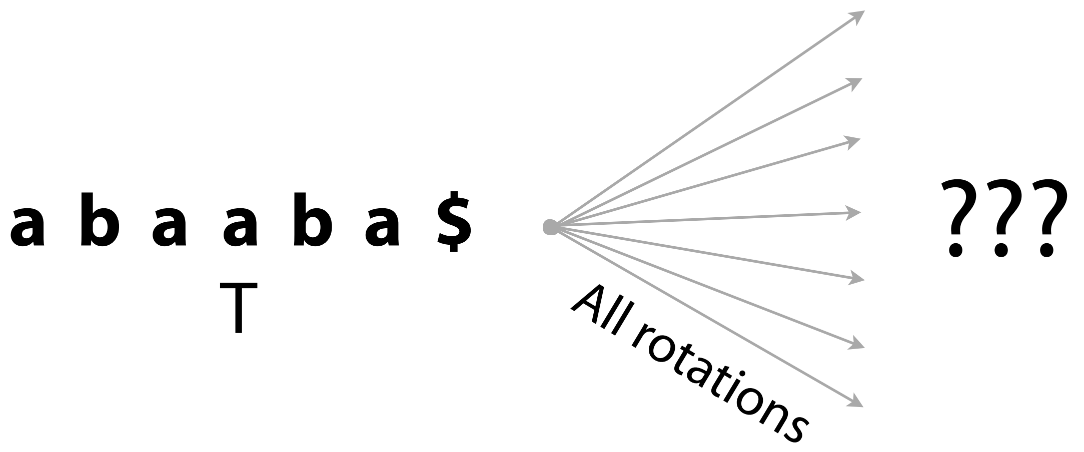
1) How to encode?

2) How to decode?

3) How is it useful for search?

Burrows-Wheeler Transform

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



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

Text rotations

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

a b c d e f $

b c d e f $ a

c d e f $ a b

d e f $ a b c

e f $ a b c d

f $ a b c d e

$ a b c d e f

(after this they repeat)

Text Rotations

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

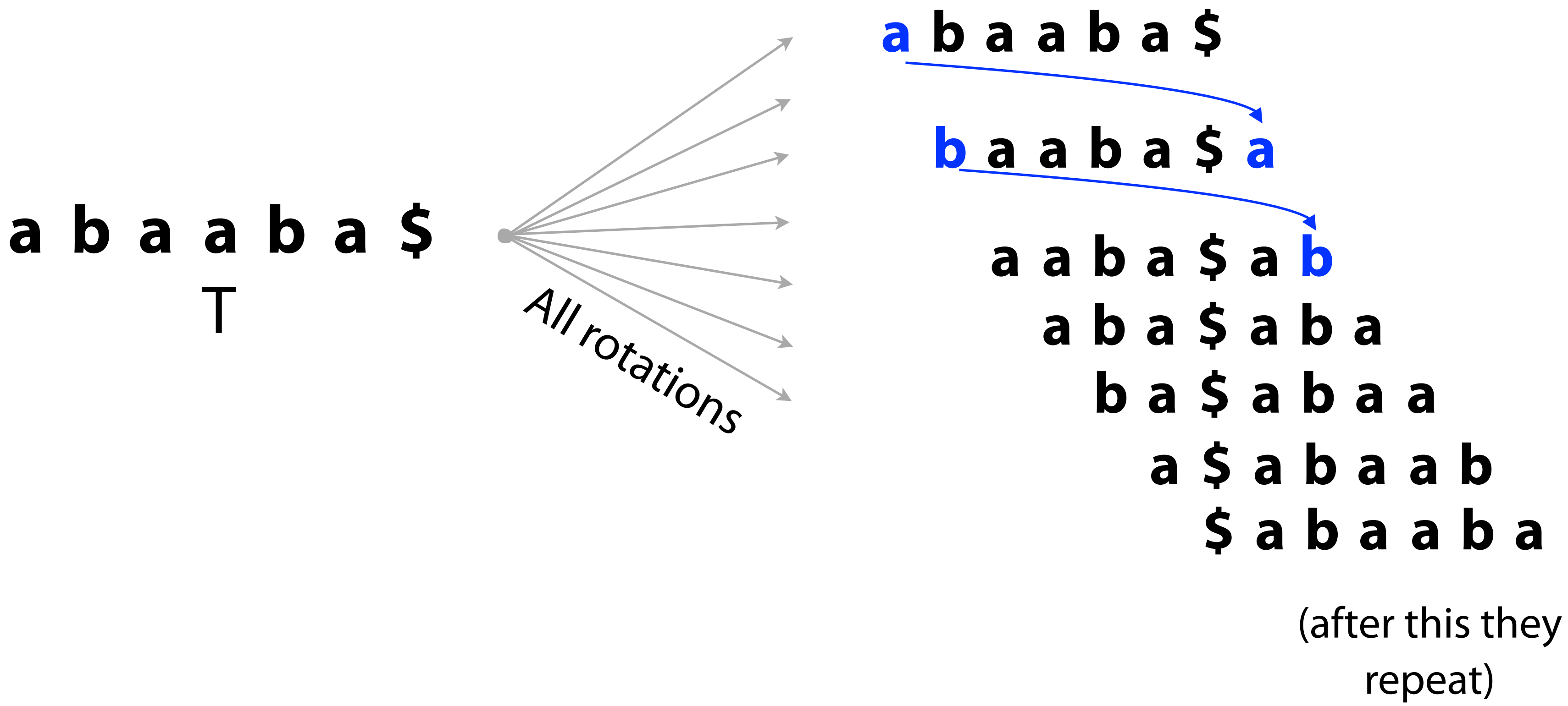
B) BACD

C) DCAB

D) CDAB

Burrows-Wheeler Transform

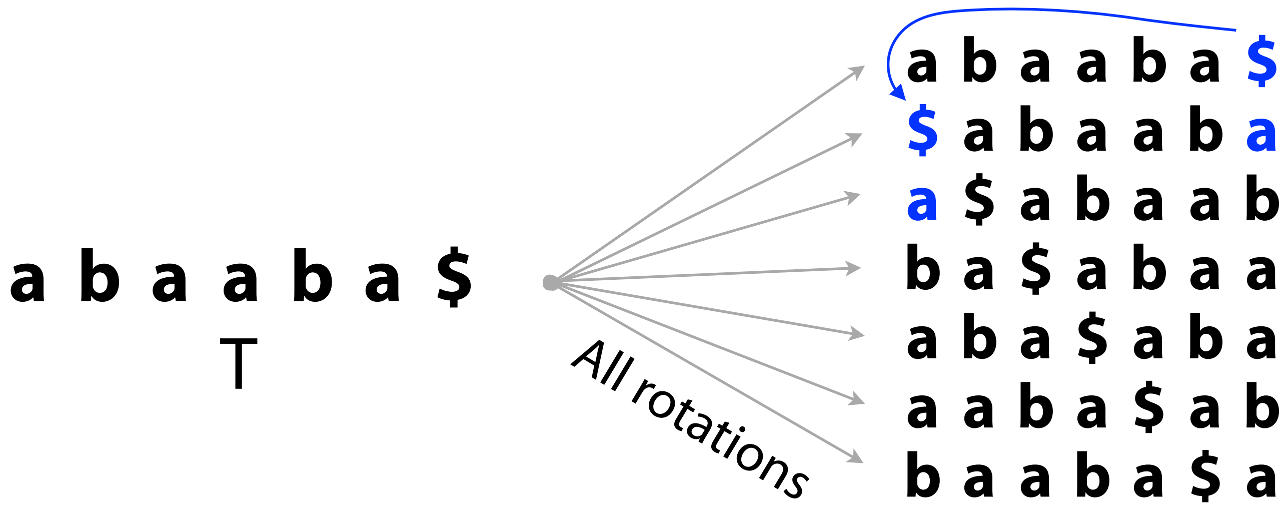
1) Build all **text rotations** of the input string



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

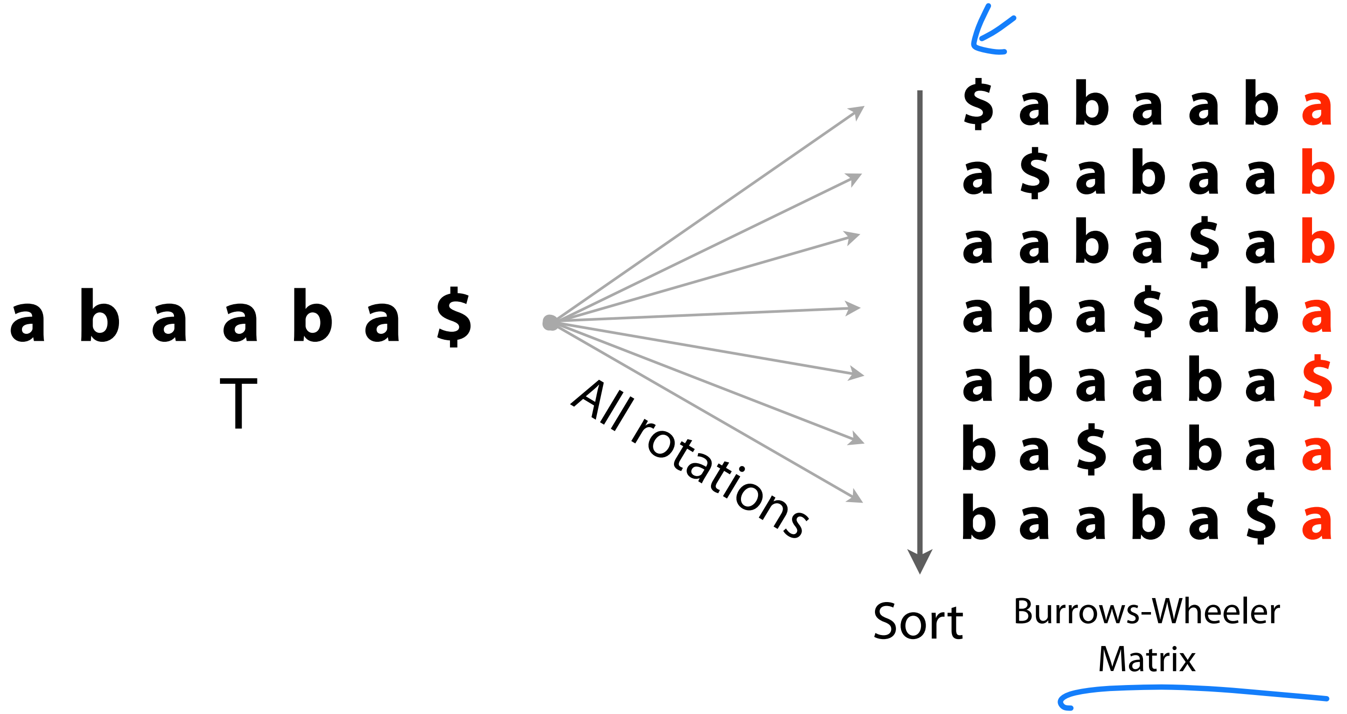
1) Build all **text rotations** of the input string



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

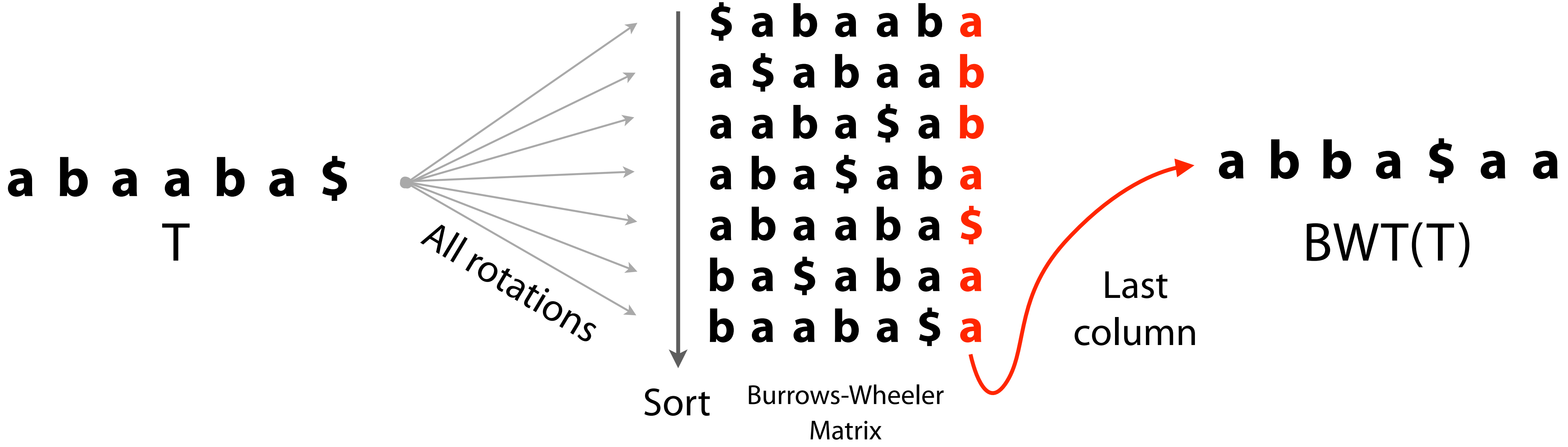
2) Sort all **text rotations** of the input string lexicographically



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

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



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

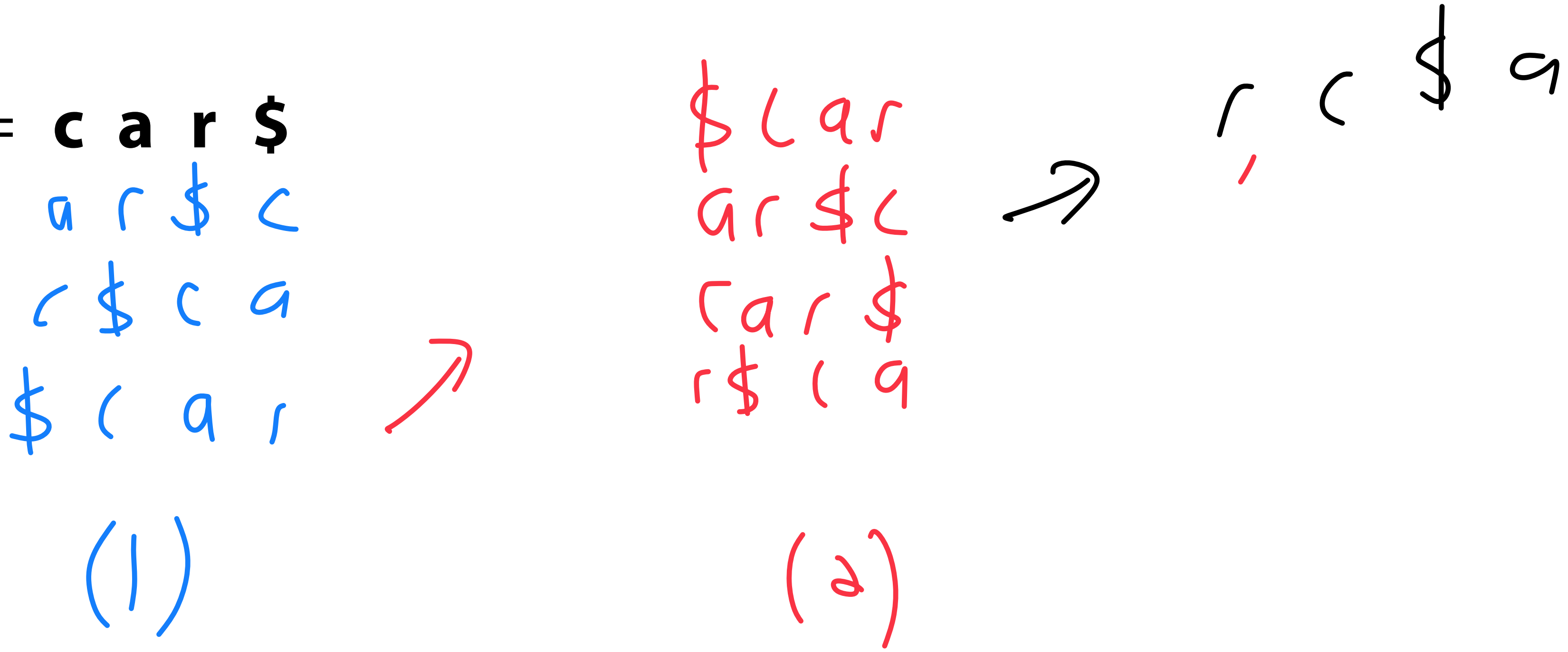
Burrows-Wheeler Transform

(1) Build all rotations

(2) Sort all rotations

(3) Take last column

T = **c a r $**

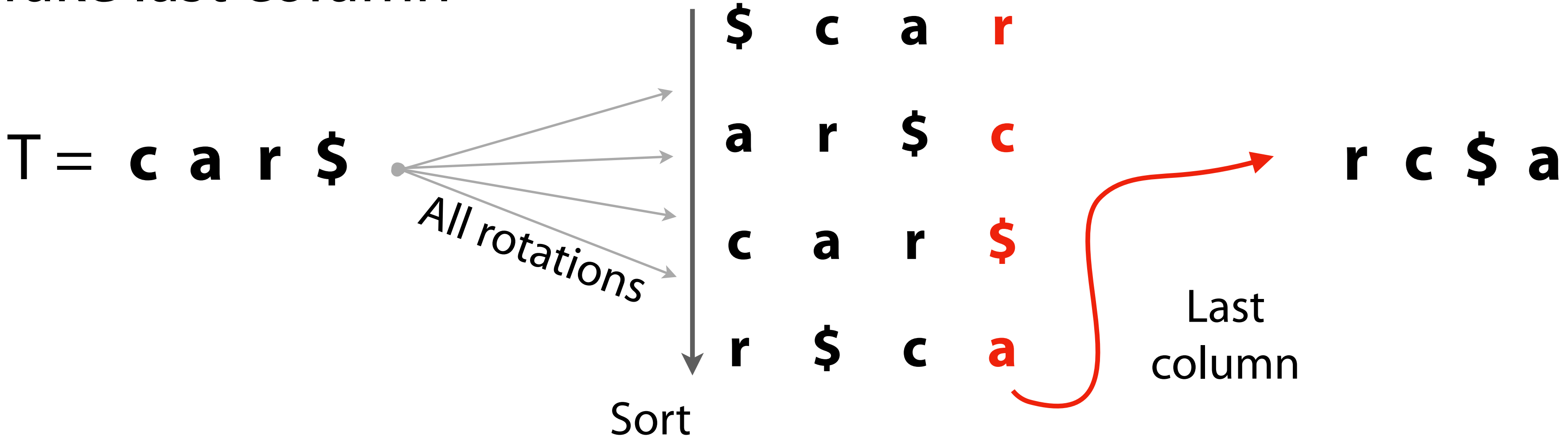


Burrows-Wheeler Transform

(1) Build all rotations

(2) Sort all rotations

(3) Take last column



Assignment 8: a_bwt

Learning Objective:

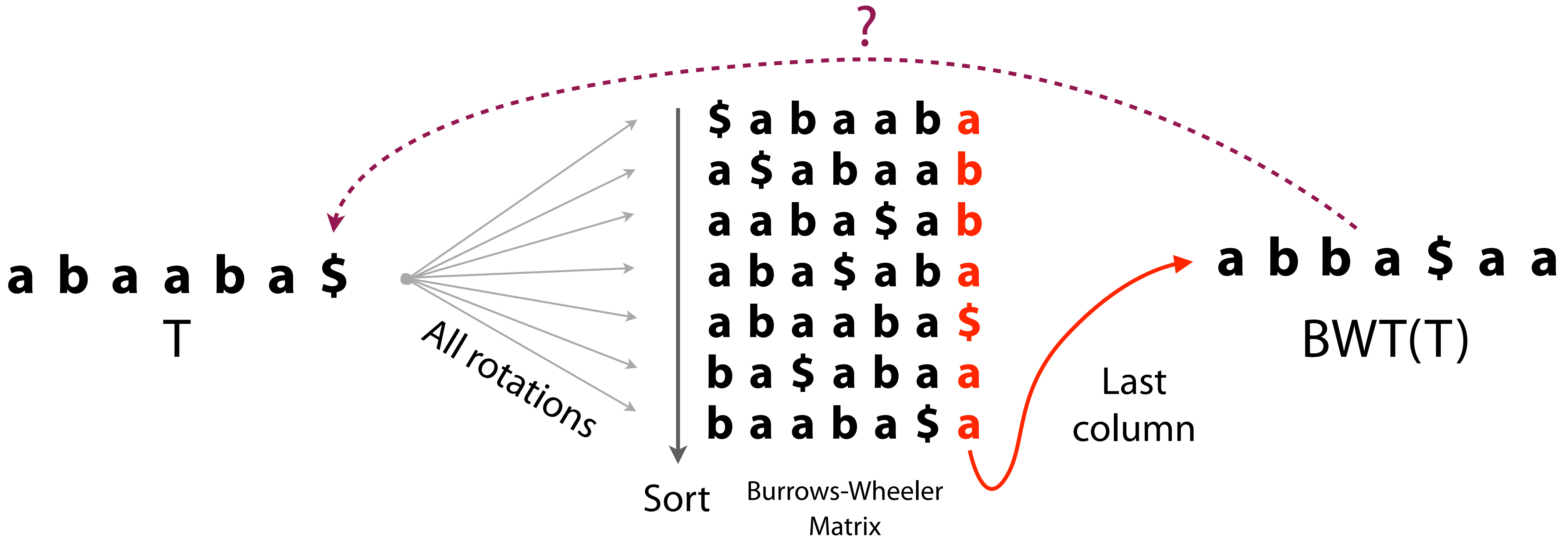
Implement the Burrows-Wheeler Transform on text

Reverse the Burrows-Wheeler Transform to reproduce text

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

Burrows-Wheeler Transform

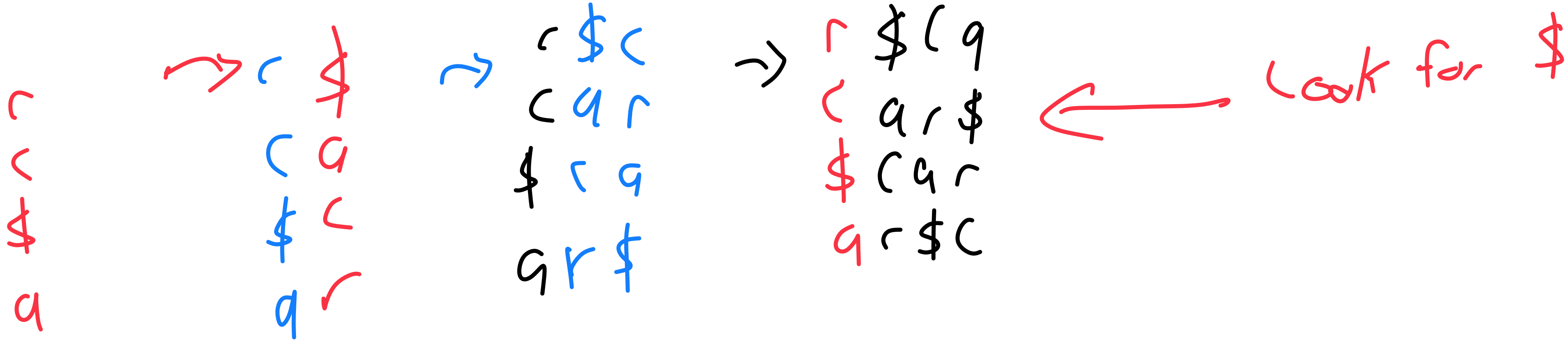
How to reverse the BWT?



Burrows-Wheeler Transform

1) prepend BWT as col
2) sort
3) Repeat

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



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 '$'

$
a
c
r

$	**c**
a	**r**
c	**a**
r	**$**

$	**c**	**a**
a	**r**	**$**
c	**a**	**r**
r	**$**	**c**

$	**c**	**a**	**r**
a	**r**	**$**	**c**
c	**a**	**r**	**$**
r	**$**	**c**	**a**

Burrows-Wheeler Transform

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

T = **c a r $**

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

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

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

…

$	c	a	**r**
a	r	$	**c**
c	a	r	**$**
r	$	c	**a**

r	**$**
c	**a**
$	**c**
a	**r**

Burrows-Wheeler Transform

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

T = **c a r $**

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**

c	**$**	**c**
c	**a**	**r**
$	**c**	**a**
a	**r**	**$**

Burrows-Wheeler Transform

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

T = **c a r $**

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

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

'c' has right context '**a**r$'.

…

$	c	a	**r**
a	r	$	**c**
c	a	r	**$**
r	$	c	**a**

$	**c**	**a**
a	**r**	**$**
c	**a**	**r**
r	**$**	**c**

Burrows-Wheeler Transform

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

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 0

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*
$\$$	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*

Burrows-Wheeler Transform

BWM with T-ranking:

F						*L*
$\$$	a_0	b_0	a_1	a_2	b_1	$\mathbf{a}_3$
$\mathbf{a}_3$	$\$$	a_0	b_0	a_1	a_2	b_1
$\mathbf{a}_1$	a_2	b_1	a_3	$\$$	a_0	b_0
$\mathbf{a}_2$	b_1	a_3	$\$$	a_0	b_0	$\mathbf{a}_1$
$\mathbf{a}_0$	b_0	a_1	a_2	b_1	a_3	$\$$
b_1	a_3	$\$$	a_0	b_0	a_1	$\mathbf{a}_2$
b_0	a_1	a_2	b_1	a_3	$\$$	$\mathbf{a}_0$

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

Burrows-Wheeler Transform

BWM with T-ranking:

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

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

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

Burrows-Wheeler Transform

BWM with T-ranking:

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

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

Burrows-Wheeler Transform: 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?

Right context:

a_1 a b a $ a b

```
$ 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
```

Right context:

a_1 a b a $ a b

These characters have the same right contexts!

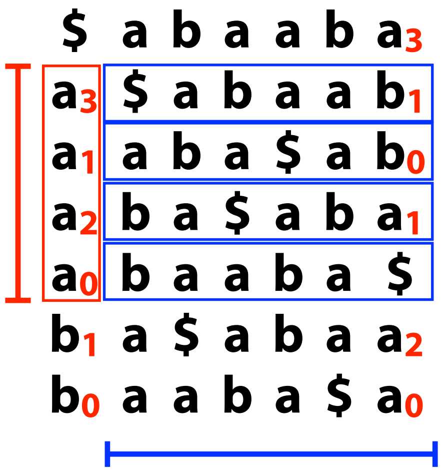
These characters *are the same character!*

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

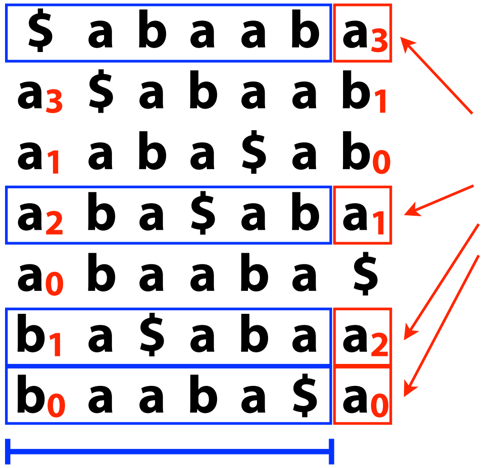
Burrows-Wheeler Transform: LF Mapping

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 = $a_3\ b_1\ b_0\ a_1\ \$\ a_2\ a_0$

What is L? The BWT

What is F? Lexicographically sorted Ordered in same #

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

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 **$**: $\mathbf{a}_3$

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

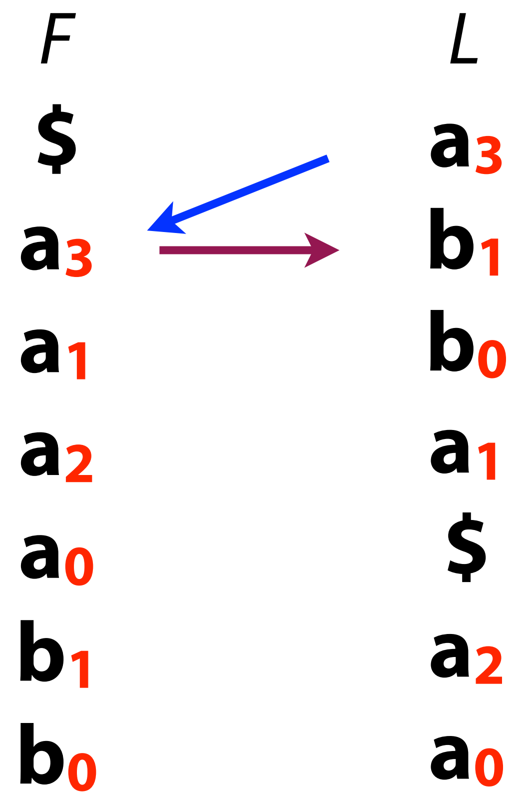
a_3 $

Burrows-Wheeler Transform: LF Mapping

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

Start in first row. *F* must have **$**.
L contains character just prior to **$**: $\mathbf{a}_3$

Jump to row *beginning* with $\mathbf{a}_3$.
L contains character just prior to $\mathbf{a}_3$: $\mathbf{b}_1$.



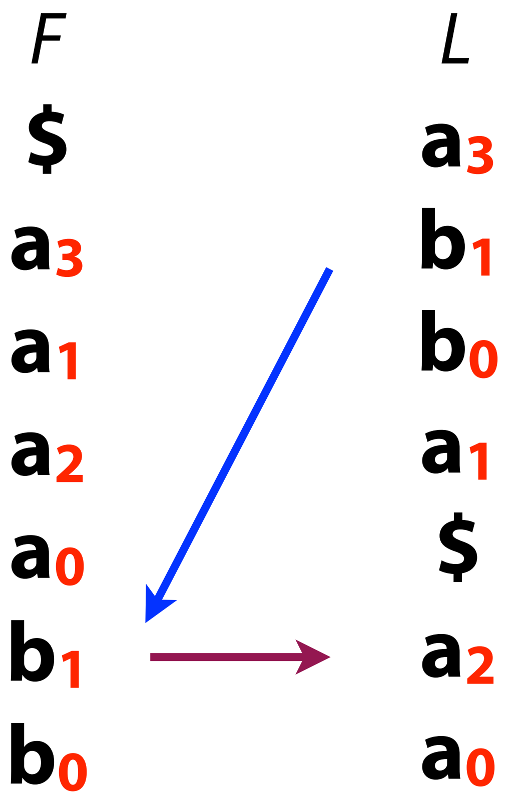
Burrows-Wheeler Transform: LF Mapping

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

Start in first row. *F* must have **$**.
L contains character just prior to **$**: $\mathbf{a}_3$

Jump to row *beginning* with $\mathbf{a}_3$.
L contains character just prior to $\mathbf{a}_3$: $\mathbf{b}_1$.

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



Burrows-Wheeler Transform: LF Mapping

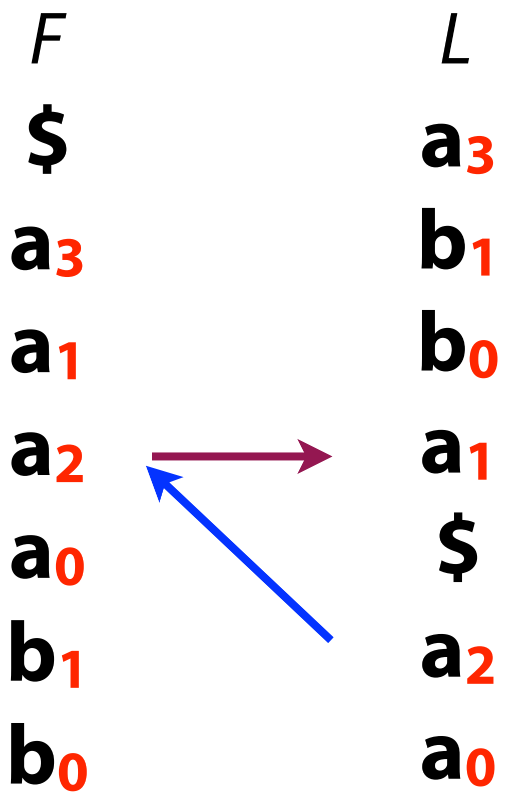
LF Mapping can be used to recover our original text too!

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 **$**.
L contains character just prior to **$**: $\mathbf{a}_3$

Jump to row *beginning* with $\mathbf{a}_3$.
L contains character just prior to $\mathbf{a}_3$: $\mathbf{b}_1$.

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

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

Repeat for $\mathbf{a}_1$, get $\mathbf{b}_0$

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

Burrows-Wheeler Transform: LF Mapping

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

Start in first row. *F* must have **$**.
L contains character just prior to **$**: $\mathbf{a}_3$

Jump to row *beginning* with $\mathbf{a}_3$.
L contains character just prior to $\mathbf{a}_3$: $\mathbf{b}_1$.

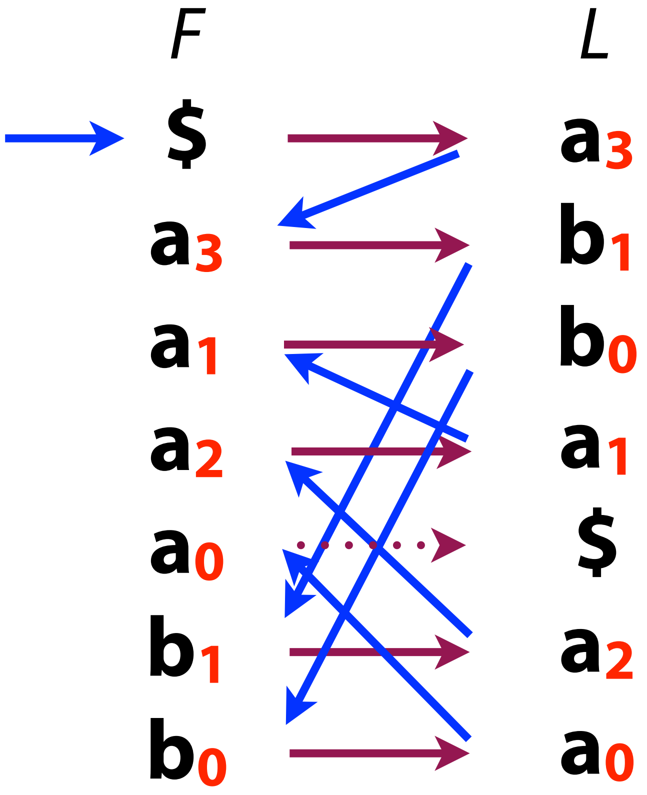
Repeat for $\mathbf{b}_1$, get $\mathbf{a}_2$

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

Repeat for $\mathbf{a}_1$, get $\mathbf{b}_0$

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

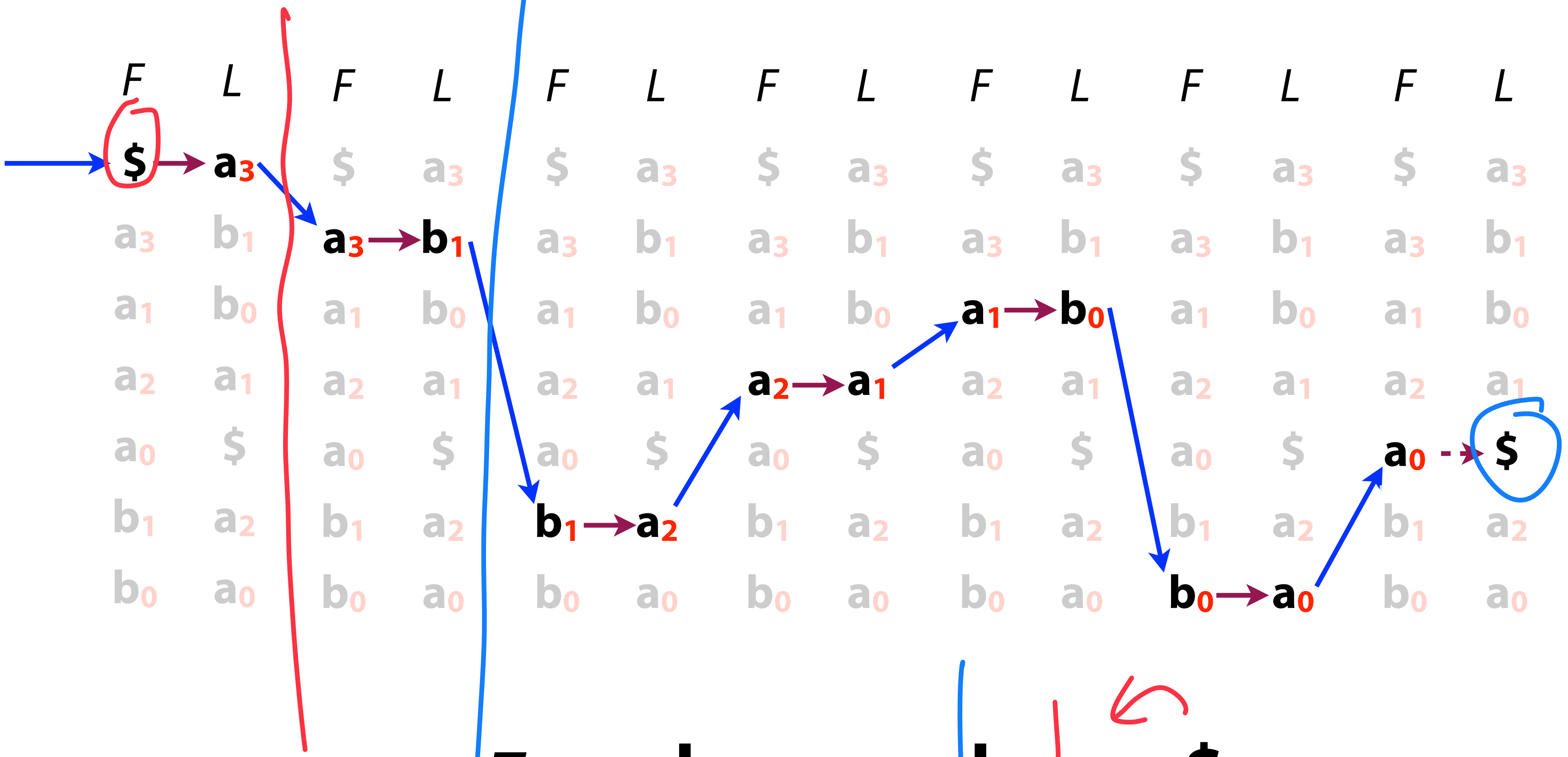
Repeat for $\mathbf{a}_0$, get **$** (done)



In reverse order, T = $\mathbf{a}_0\ \mathbf{b}_0\ \mathbf{a}_1\ \mathbf{a}_2\ \mathbf{b}_1\ \mathbf{a}_3$ **$**

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

F-Rank: Order by F

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

What is good about F-rank?

Burrows-Wheeler Transform: A better ranking

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

→ 1 A, 2 B, 2 C, 1 D

What is the BWM index for my first instance of C? **(C_0)** [0-base for answer]

1 $
1 A
2 B

→ 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? **(C_0)** [0-base for answer]

	F						*L*
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? **(C_0)** [0-base for answer]

Skip '$' (1)

Skip 'A' (1)

Skip 'B' (2)

Look-up F[**4**]

F						*L*
$	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 **A**s, 400 **C**s, 250 **G**s and 700 **T**s and **$** < **A** < **C** < **G** < **T**

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

Burrows-Wheeler Transform: A better ranking

Say *T* has 300 **A**s, 400 **C**s, 250 **G**s and 700 **T**s and **$** < **A** < **C** < **G** < **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)

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

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

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:

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**

L is the same size as *T*

F can be represented as array of |Σ| integers (or not stored at all!)

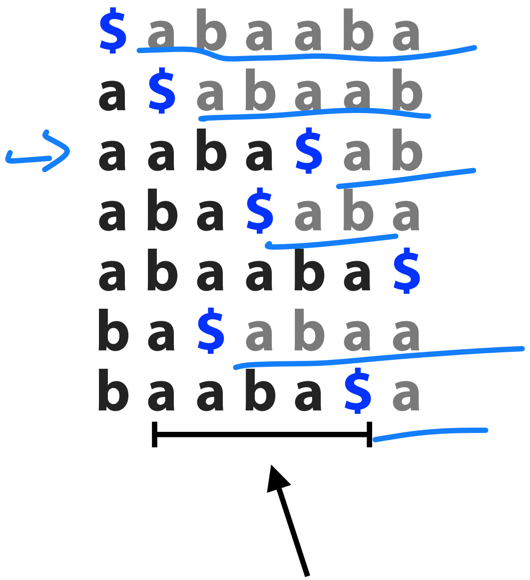
We're discarding *T* — *we can recover it from L!*

a = 4

b = 2

FM Index: Querying

Can we query like the suffix array?



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, and we don’t have T.
Binary search not possible.

FM Index: Querying

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

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)

FM Index: Querying

The BWM is a lot like the suffix array — maybe we can query 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, and we don't have T.

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 = **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**

$a_0 = 1$

$a_3 = 4$

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 = **aba**

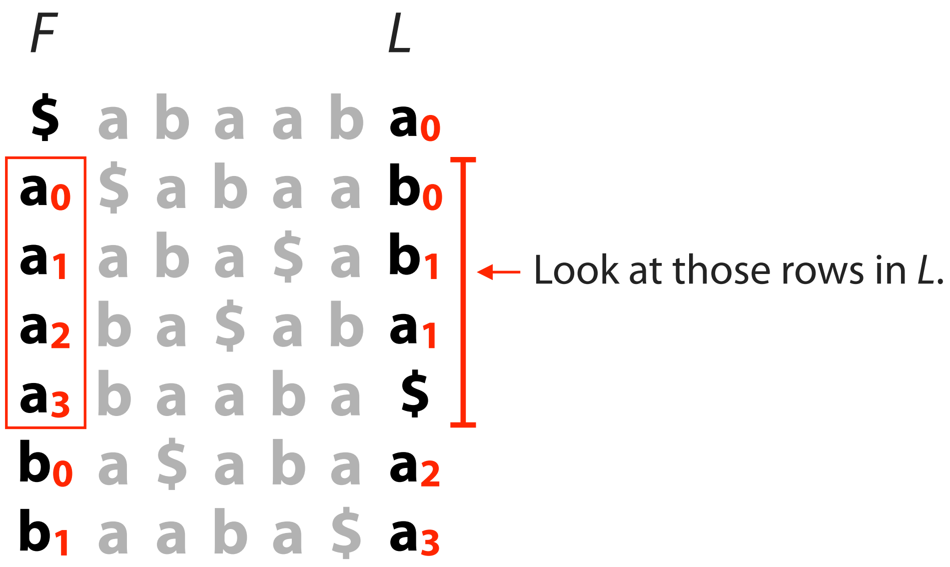
Easy to find all the rows beginning with **a**

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

FM Index: Querying

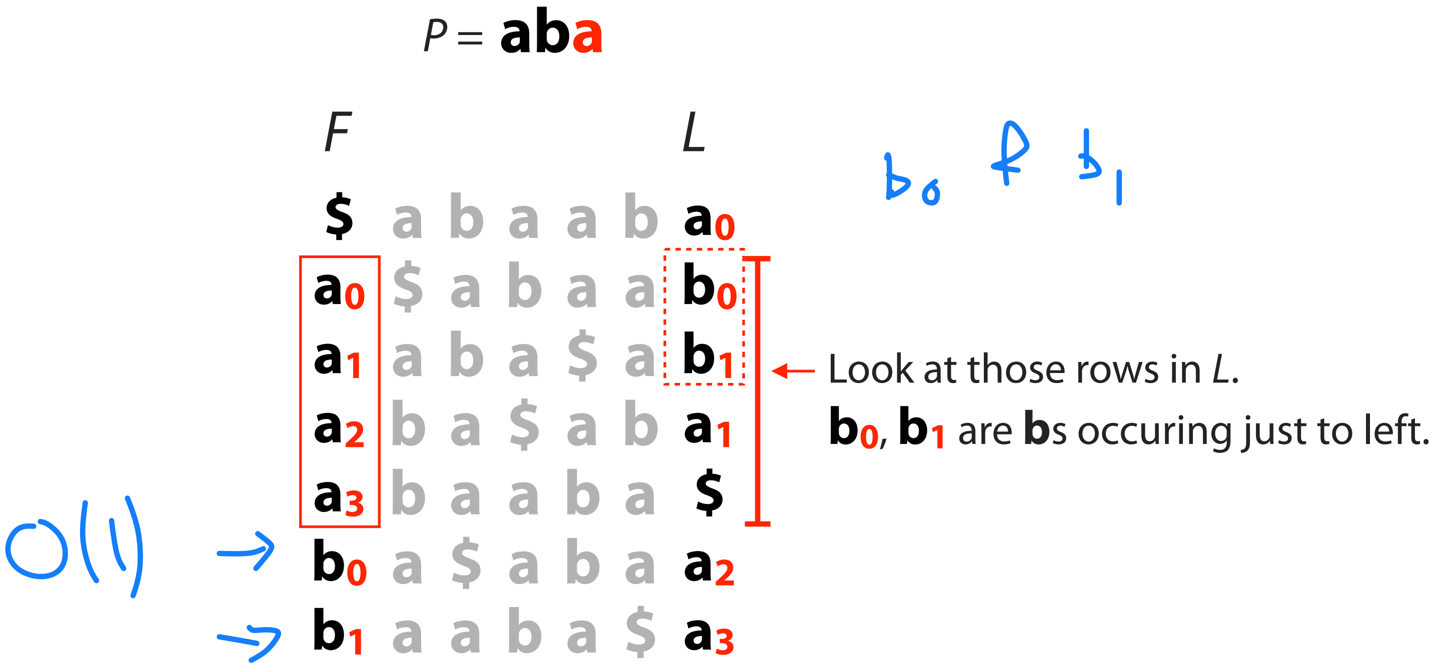
We have rows beginning with **a**, now we want rows beginning with **ba**

P = **aba**



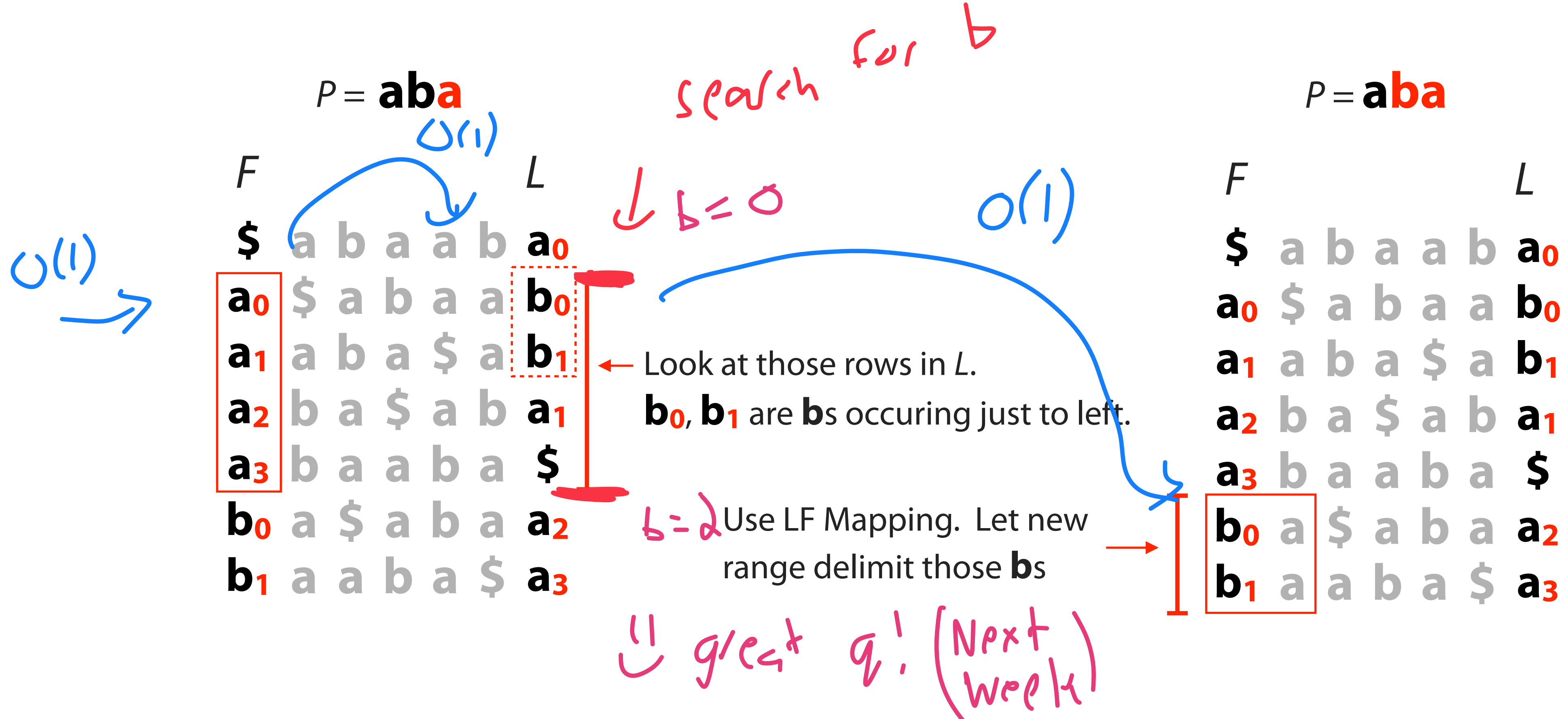
FM Index: Querying

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



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**

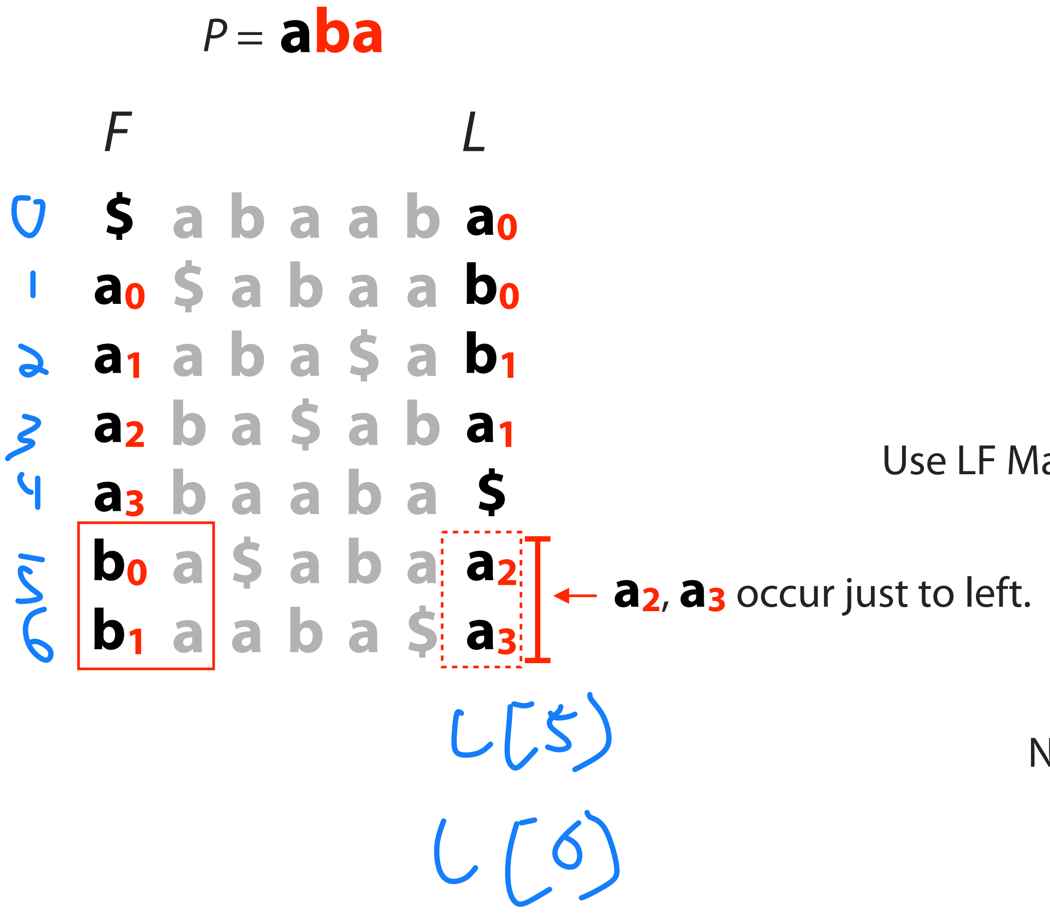
P = **aba**

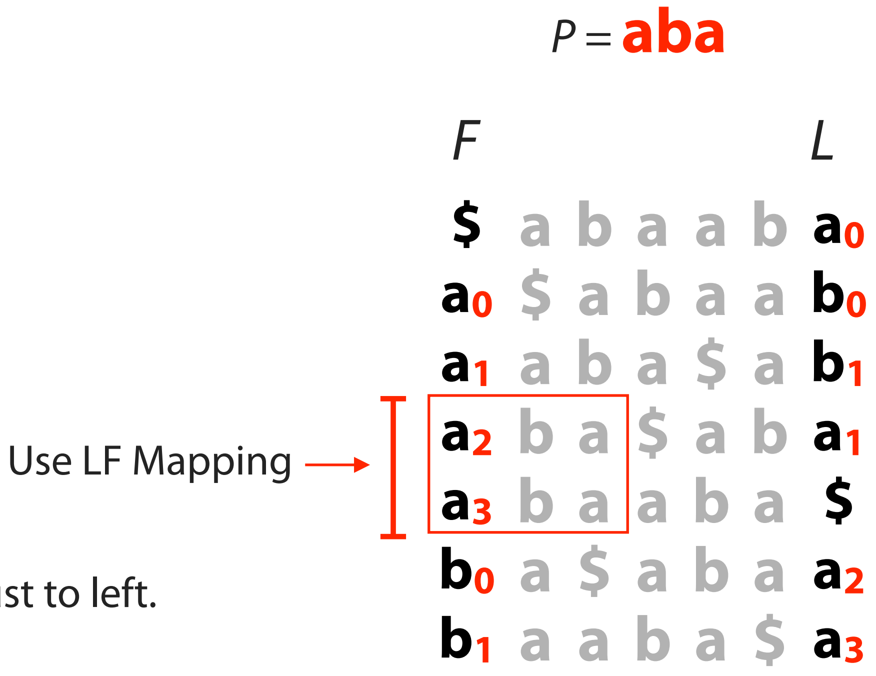
F						*L*
\$	a	b	a	a	b	a_0
a_0	\$	a	b	a	a	b_0
a_1	a	b	a	\$	a	b_1
a_2	b	a	\$	a	b	a_1
a_3	b	a	a	b	a	\$
b_0	a	\$	a	b	a	a_2
b_1	a	a	b	a	\$	a_3

← a_2, a_3 occur just to left.

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 = **bba**

F						*L*	
$	a	b	a	a	b	a_0	
a_0	$	a	b	a	a	b_0	
a_1	a	b	a	$	a	b_1	
a_2	b	a	$	a	b	a_1	
a_3	b	a	a	b	a	**$**	
b_0	a	$	a	b	a	a_2	← No **b**s!
b_1	a	a	b	a	$	a_3	

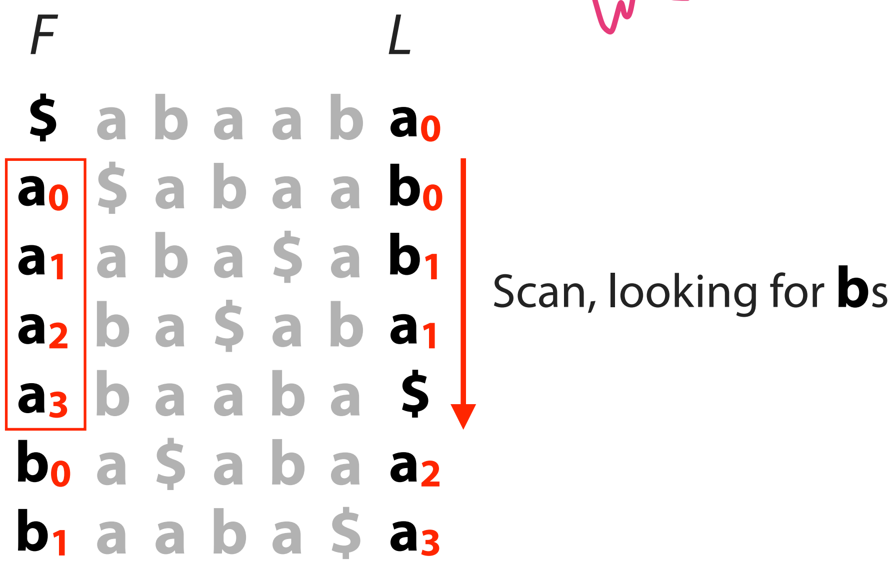
Rows with **ba** prefix: b_0, b_1

FM Index: Querying

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

P = **aba**

we just did in pink :)

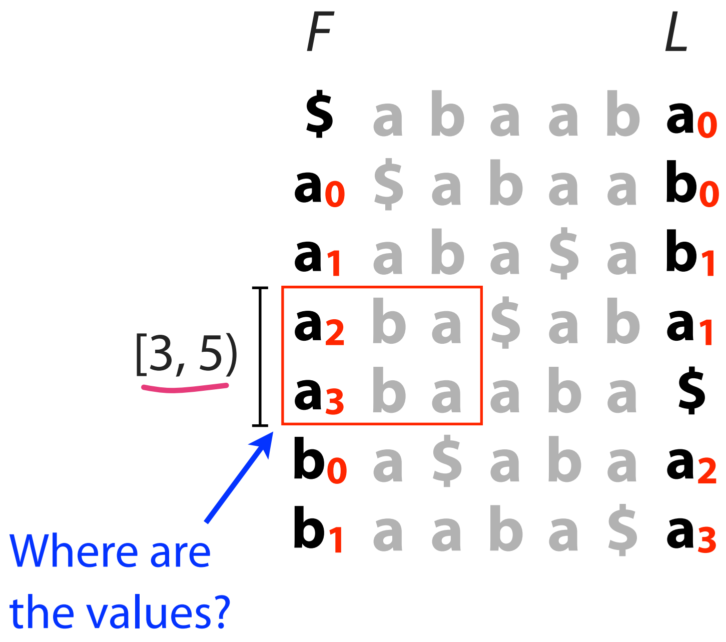


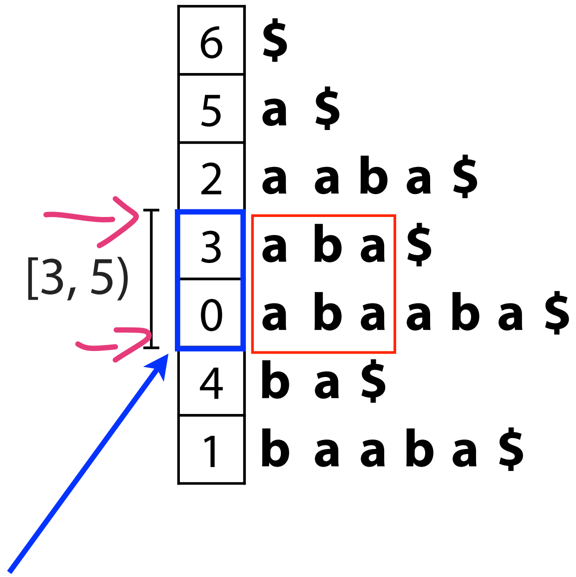
FM Index: Querying

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

$P =$ **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

T		BWT(T)
B A N A N A $	⟷	A N N B $ A A

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$wwdd__nnooooaattTmmmrrrrrooo__ooo

It_was_the_best_of_times_it_was_the_worst_of_times$

s$esttssfftteww_hhmmbootttt_ii__woeeaaressIi_______

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

Burrows-Wheeler Transform

F

```
orrow_and_tomorrow_and_tomorrow$tom
ow$tomorrow_and_tomorrow_and_tomorr
ow_and_tomorrow$tomorrow_and_tomorr
ow_and_tomorrow_and_tomorrow$tomorr
row$tomorrow_and_tomorrow_and_tomor
row_and_tomorrow$tomorrow_and_tomor
row_and_tomorrow_and_tomorrow$tomor
rrow$tomorrow_and_tomorrow_and_tomo
```

prior character to F

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

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)

$ 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)

Suffix Array is O(m)

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)

a
b
b
a
$
a
a

BWT(T)

Suffix Array is O(m)

BWT is O(m)

The BWT has a better constant factor! & is better compressable

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"

a b a a b a $

SA	Suffix	BWT	
6	$	a	T[5]
5	a $	b	T[4]
2	a a b a $	b	T[1]
3	a b a $	a	T[2]
0	a b a a b a $	$	SA[0]
4	b a $	a	T[3]
1	b a a b a $	a	T[0]