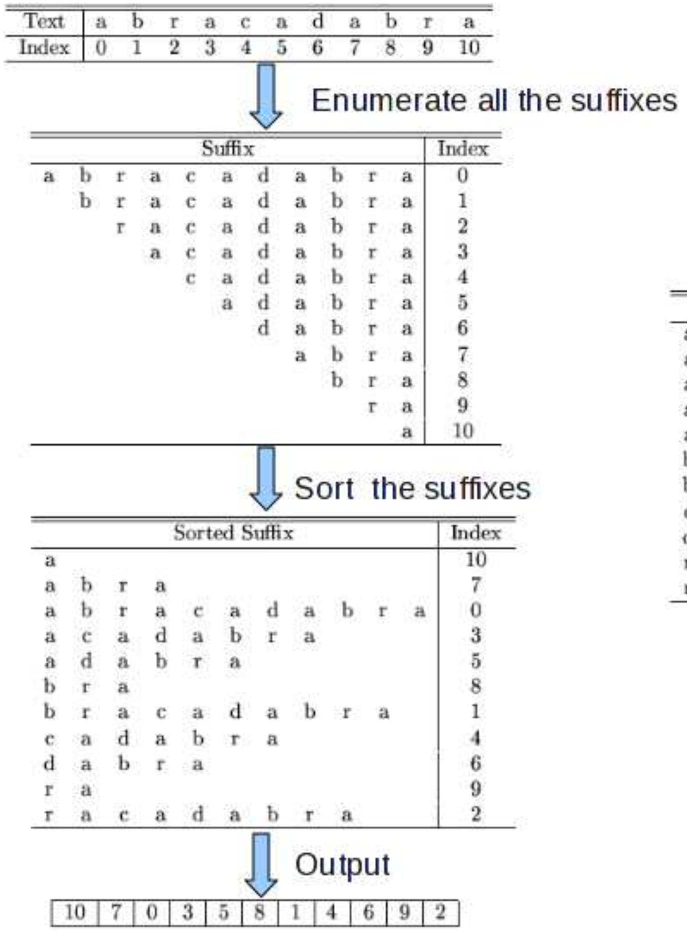
Suffix Array

A suffix array is a data structure designed for efficient searching of a large text. It is an array of integers giving the starting position of the suffixes of a string in lexicographical order.

It can be used as index to quickly locate every occurrence of a substring within the string. Finding every occurrence of the substring is equivalent to finding every suffix that begins with the substring.

Thanks to the lexicographical ordering these suffixes will be grouped together in the suffix array, and can be found efficiently using a binary search.

Suffix Array



Text	a	b	r	a	c	a	d	a	b	r	a
Index	0	1	2	3	4	5	6	7	8	9	10

Enumerate all the suffixes

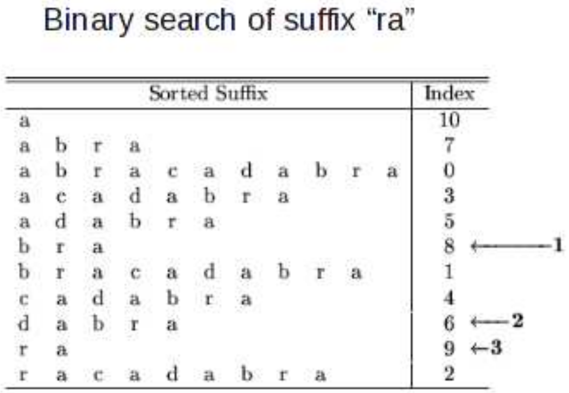
Suffix	Index
a b r a c a d a b r a	0
b r a c a d a b r a	1
r a c a d a b r a	2
a c a d a b r a	3
c a d a b r a	4
a d a b r a	5
d a b r a	6
a b r a	7
b r a	8
r a	9
a	10

Sort the suffixes

Sorted Suffix	Index
a	10
a b r a	7
a b r a c a d a b r a	0
a c a d a b r a	3
a d a b r a	5
b r a	8
b r a c a d a b r a	1
c a d a b r a	4
d a b r a	6
r a	9
r a c a d a b r a	2

Output

10	7	0	3	5	8	1	4	6	9	2

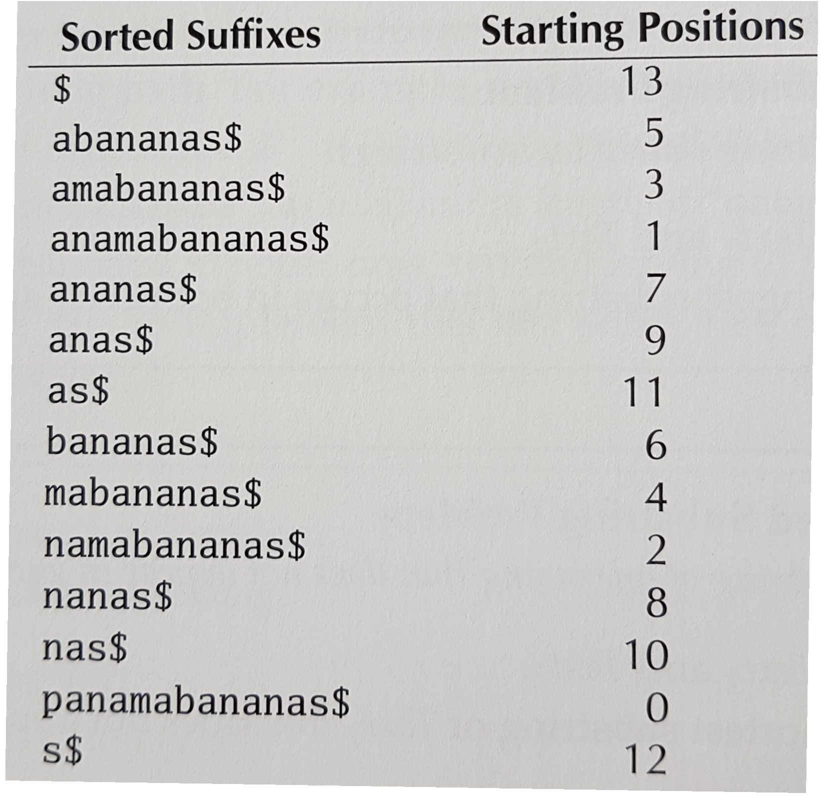


Binary search of suffix "ra"

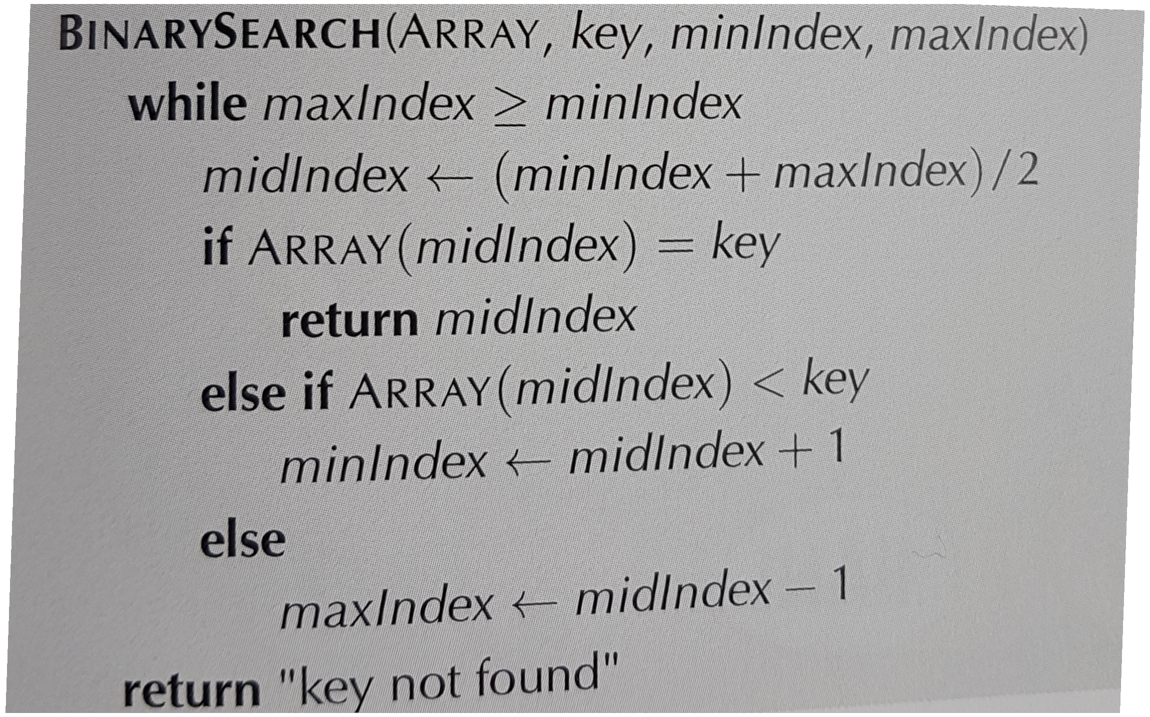
Sorted Suffix	Index	
a	10	
a b r a	7	
a b r a c a d a b r a	0	
a c a d a b r a	3	
a d a b r a	5	
b r a	8	←1
b r a c a d a b r a	1	
c a d a b r a	4	
d a b r a	6	←2
r a	9	←3
r a c a d a b r a	2	

Suffix array is a memory-efficient alternative to suffix trees

SUFFIXARRAY (“panamabananas$”) = [13,5,3,1,7,9,11,6,4,2,8,10,0,12]



Sorted Suffixes	Starting Positions
$	13
abananas$	5
amabananas$	3
anamabananas$	1
ananas$	7
anas$	9
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12



```
BINARYSEARCH(ARRAY, key, minIndex, maxIndex)
    while $maxIndex \geq minIndex$
        $midIndex \leftarrow (minIndex + maxIndex)/2$
        if ARRAY($midIndex$) = $key$
            return $midIndex$
        else if ARRAY($midIndex$) < $key$
            $minIndex \leftarrow midIndex + 1$
        else
            $maxIndex \leftarrow midIndex - 1$
    return "key not found"
```

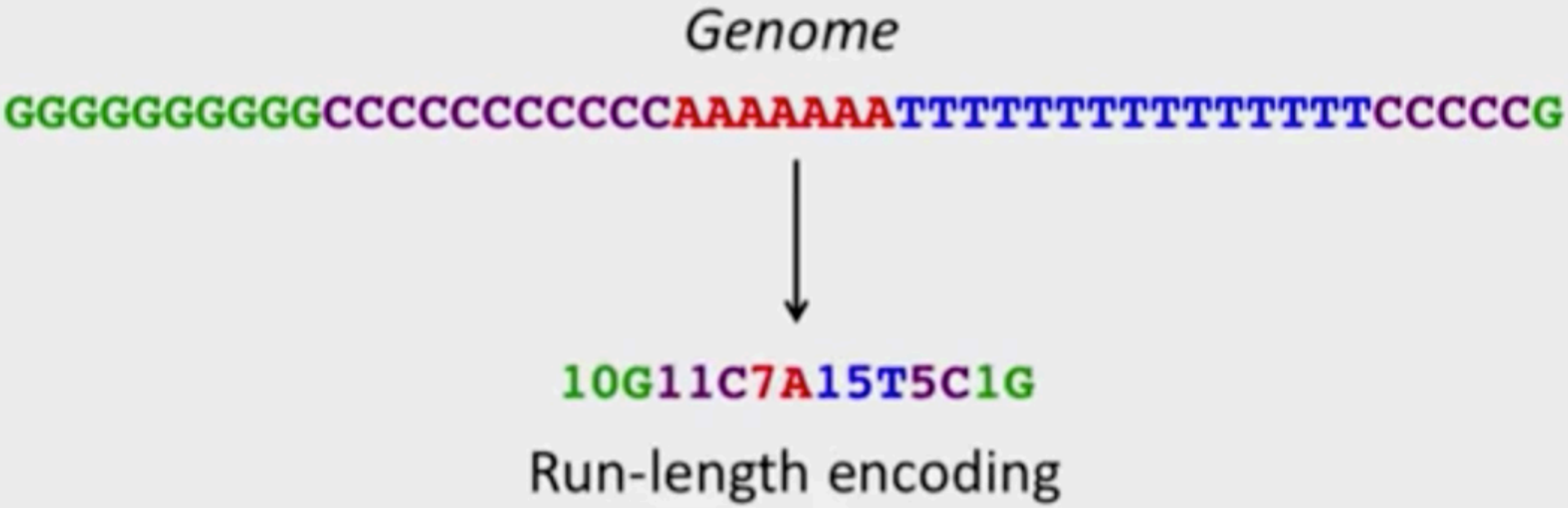
Suffix Array

How do we build a suffix array?

- Easiest solution: build a suffix tree
- Traverse the tree in DFS, lexicographically picking edges outgoing from each node and fill the suffix array
- O(n) time
- Waste of space: can we do it directly in O(n) time? Unknown until 2003

Idea #1: Run-Length Encoding

- **Run-length encoding:** compresses a run of *n* identical symbols.



Genome

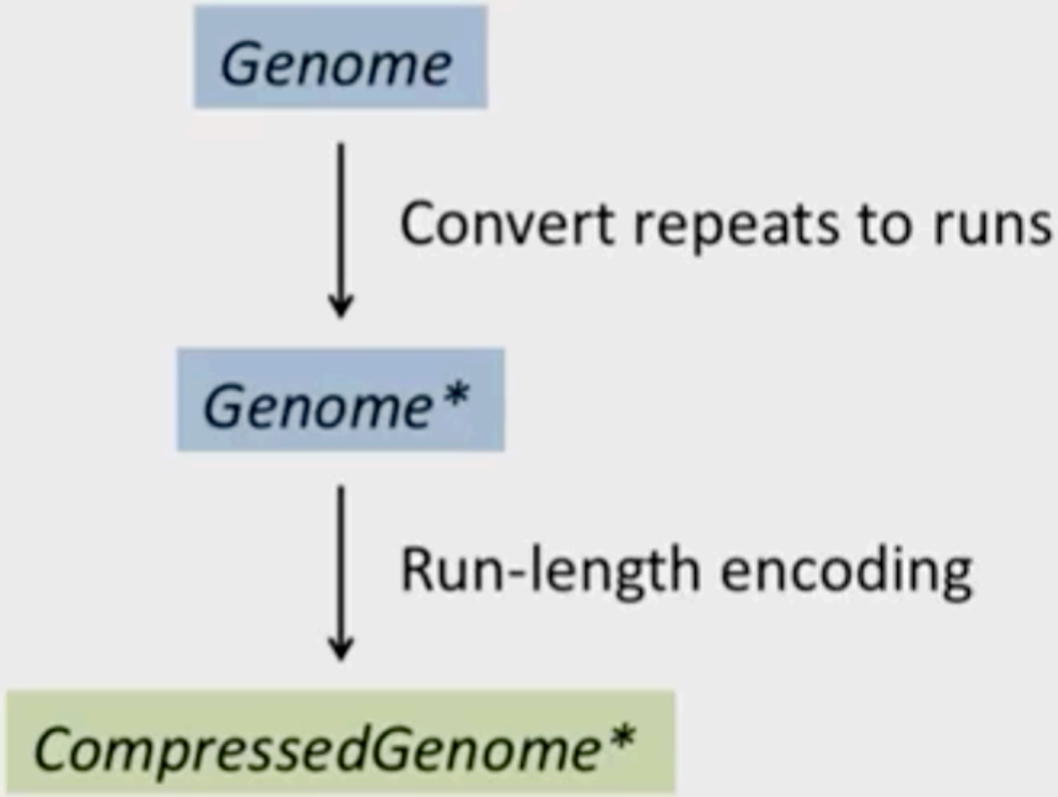
GGGGGGGGGGCCCCCCCCCCCAAAAAAATTTTTTTTTTTTTTTCCCCCG

10G11C7A15T5C1G

Run-length encoding

Converting Repeats to Runs

- …but they do have lots of repeats!



Genome

Convert repeats to runs

*Genome**

Run-length encoding

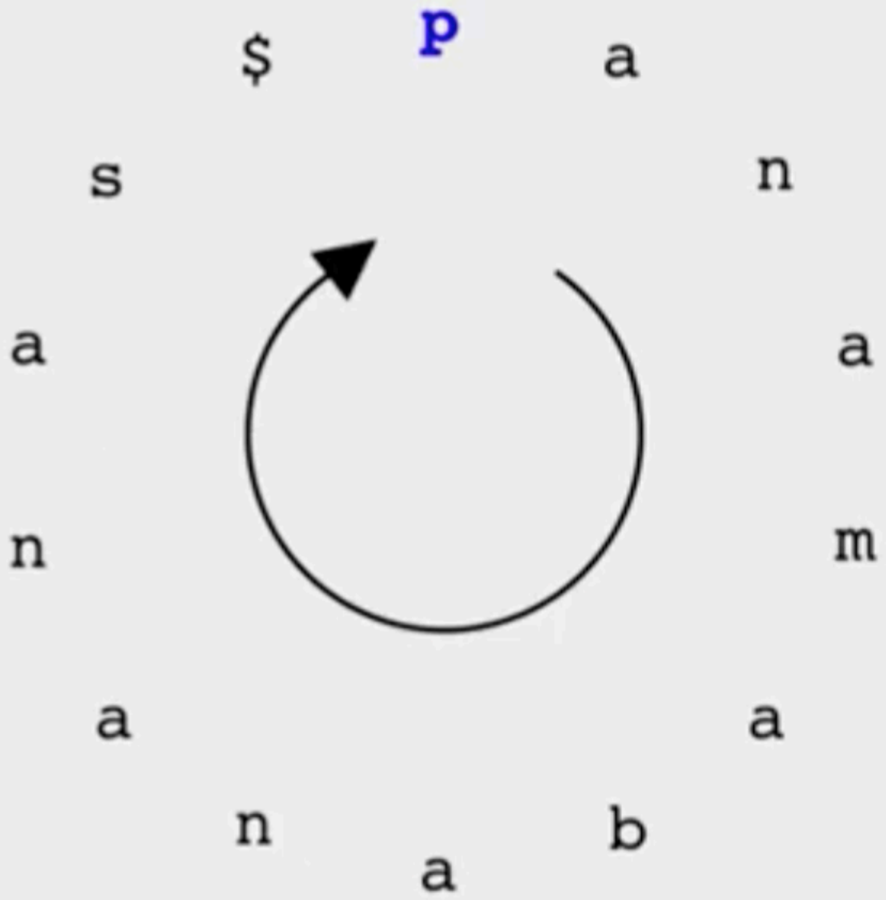
*CompressedGenome**

Burrows-Wheeler transformation

It performs a transformation of an input sequence consisting of a reversible permutation of the sequence characters which gives a new string that is **easier to compress**: if the original string has several substrings that occur often, then the transformed string has several places where a single character is repeated multiple time.

The Burrows-Wheeler Transform

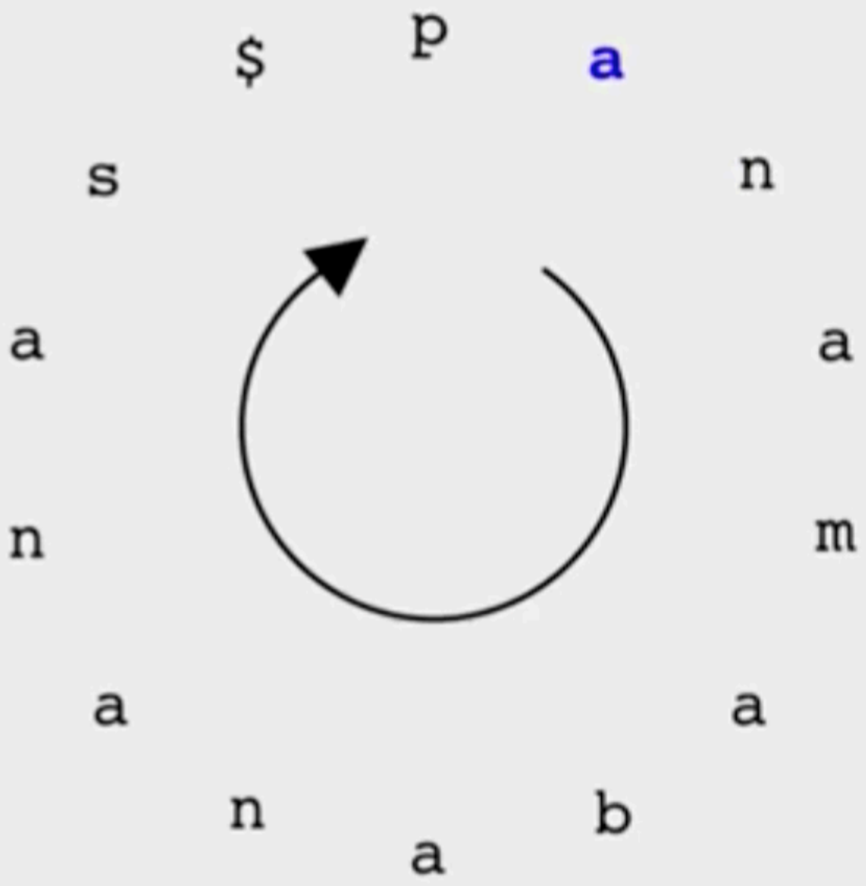
panamabananas$



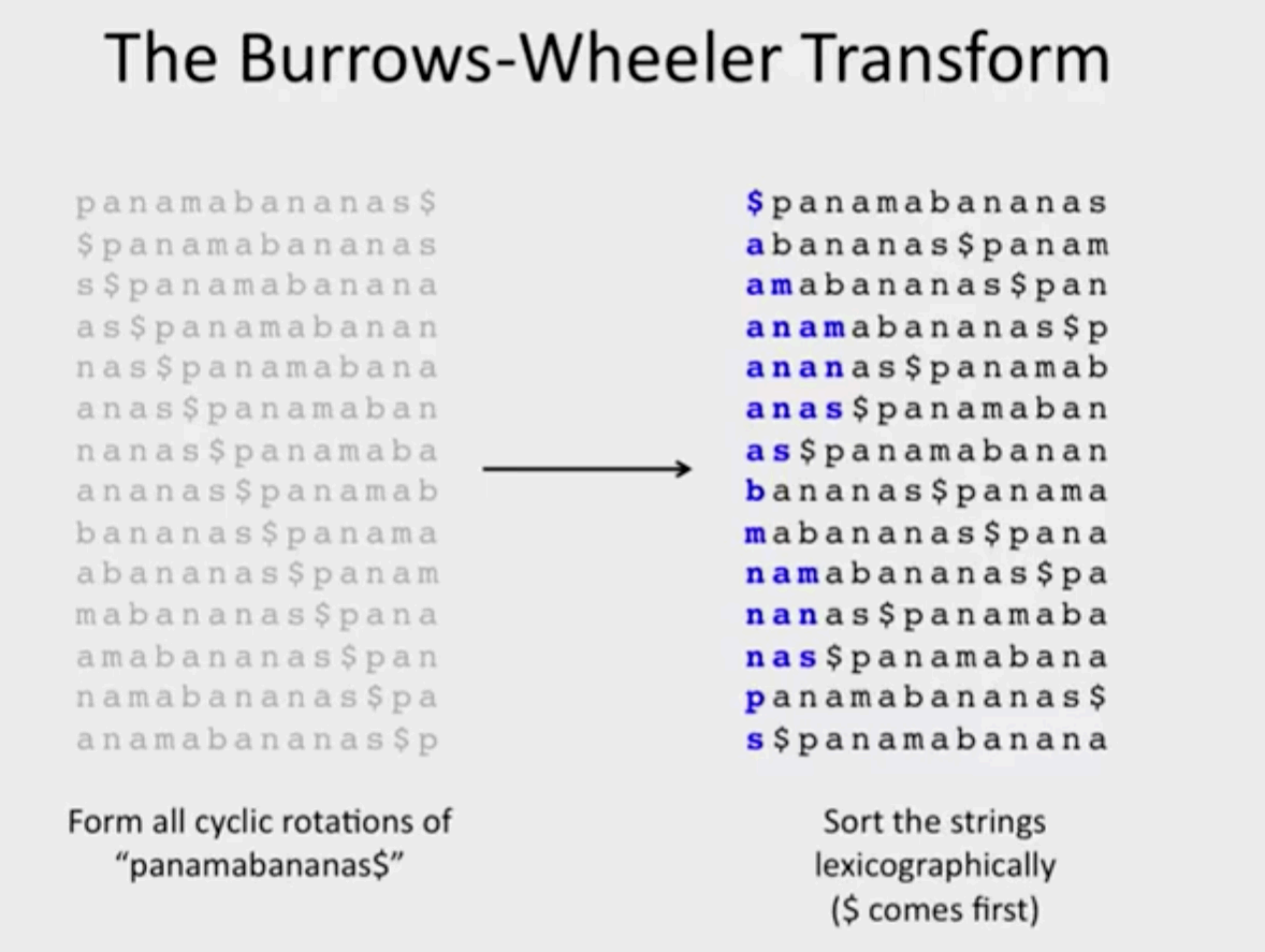
Form all cyclic rotations of
"panamabananas$"

The Burrows-Wheeler Transform

```
panamabananas$
$panamabananas
s$panamabanana
as$panamabanan
nas$panamabana
anas$panamaban
nanas$panamaba
ananas$panamab
bananas$panama
abananas$panam
mabananas$pana
amabananas$pan
namabananas$pa
anamabananas$p
```



Form all cyclic rotations of "panamabananas$"



The Burrows-Wheeler Transform

```
panamabananas$
$panamabananas
s$panamabanana
as$panamabanan
nas$panamabana
anas$panamaban
nanas$panamaba
ananas$panamab
bananas$panama
abananas$panam
mabananas$pana
amabananas$pan
namabananas$pa
anamabananas$p
```

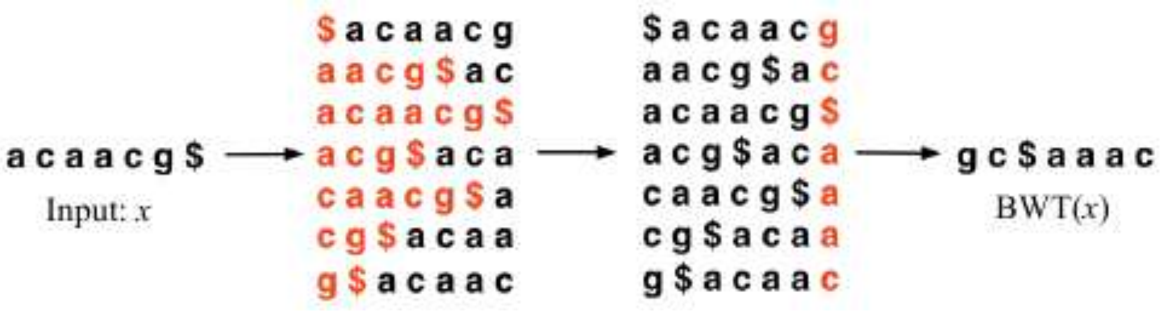
Form all cyclic rotations of
“panamabananas$”

```
$panamabananas
abananas$panam
amabananas$pan
anamabananas$p
ananas$panamab
anas$panamaban
as$panamabanan
bananas$panama
mabananas$pana
namabananas$pa
nanas$panamaba
nas$panamabana
panamabananas$
s$panamabanana
```

Sort the strings
lexicographically
($ comes first)

Burrows-Wheeler transformation

Reversible permutation used originally in lossless data compression:



Input (x) transformed in the matrix of all circular shifts of x, sorted lexicographically. Then the last column of the matrix became the BWT(x)

Another Example

appellee$

appellee$
ppellee$a
pellee$ap
ellee$app
llee$appe
lee$appel
ee$appell
e$appelle
$appellee

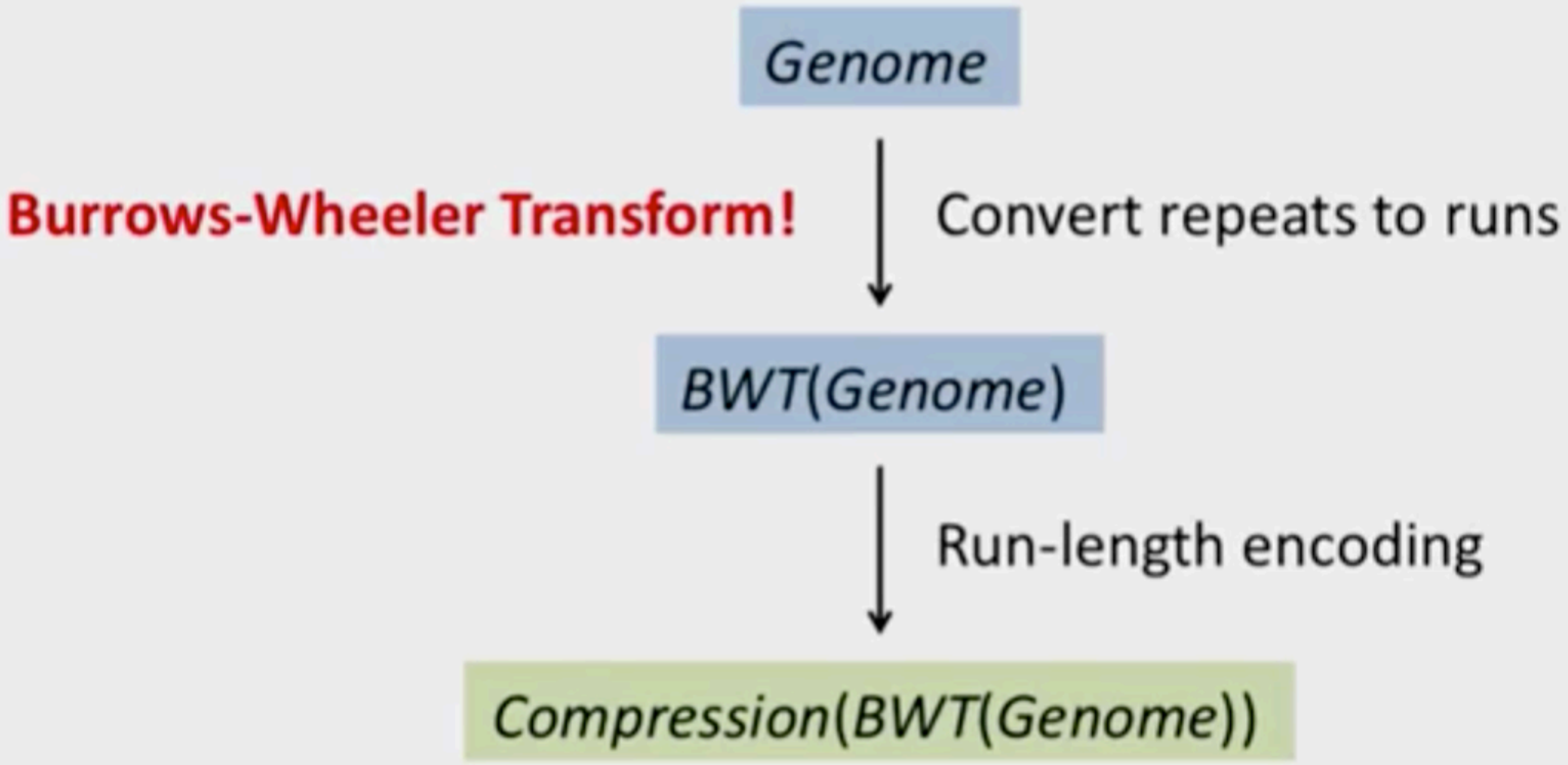
sort →

$appellee
appellee$
e$appelle
ee$appell
ellee$app
lee$appel
llee$appe
pellee$ap
ppellee$a

BWT(appellee$) =
e$elplepa

Doesn’t always improve the compressibility...

BWT: Converting Repeats to Runs



Genome

Burrows-Wheeler Transform!

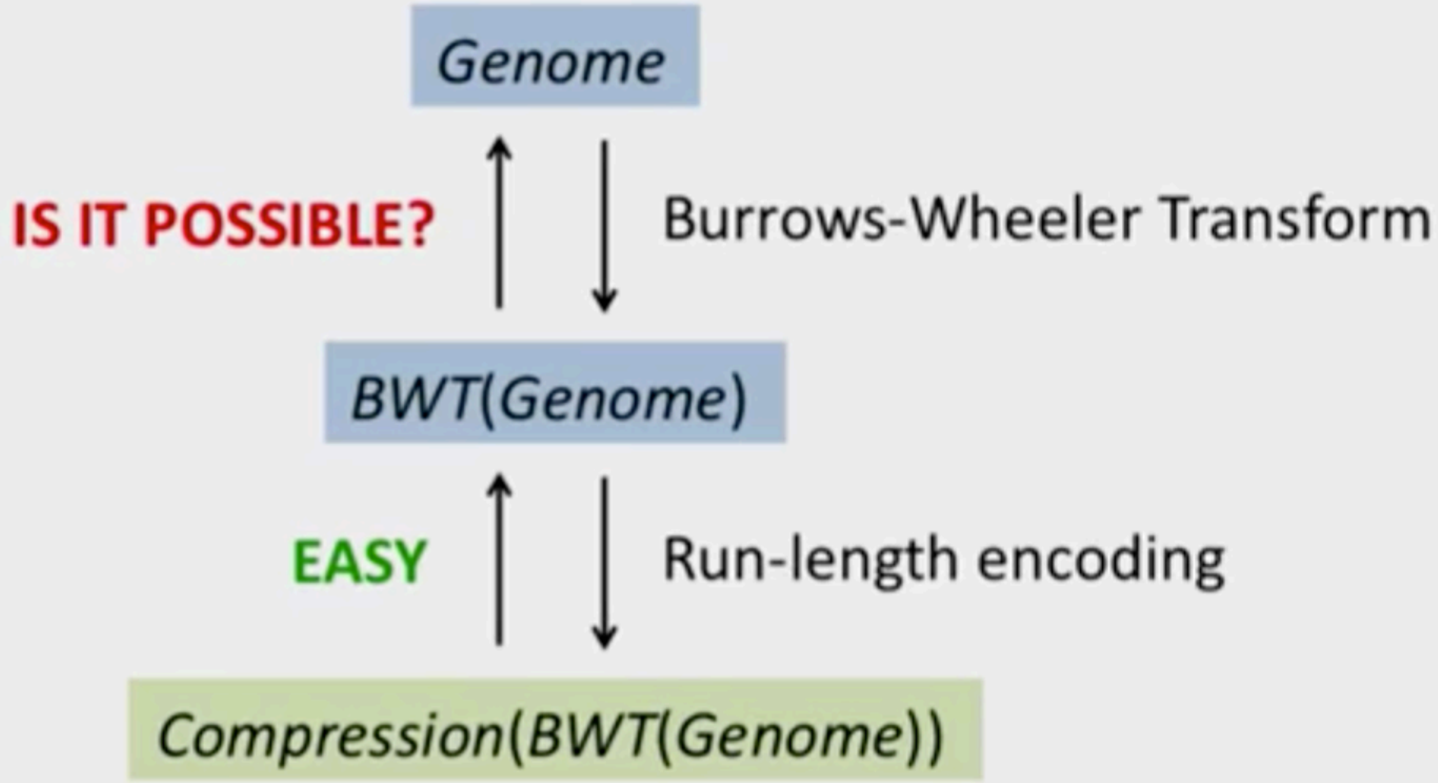
Convert repeats to runs

BWT(*Genome*)

Run-length encoding

Compression(*BWT*(*Genome*))

How Can We Decompress?



Genome

IS IT POSSIBLE?

Burrows-Wheeler Transform

BWT(*Genome*)

EASY

Run-length encoding

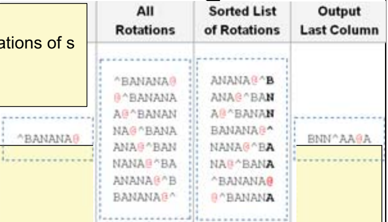
Compression(*BWT*(*Genome*))

Burrows-Wheeler Transform (BWT)

http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf

- Transform: ^BANANA@ INTO: BNN^AA@A

```
function BWT (string s)
    create a table, rows are all possible rotations of s
    sort rows alphabetically
    return (last column of the table)
```

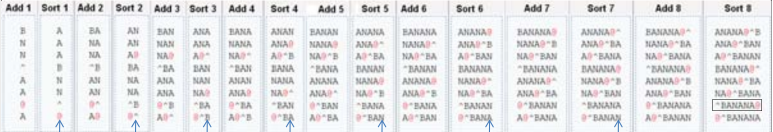


	All Rotations	Sorted List of Rotations	Output Last Column
^BANANA@	^BANANA@	ANANA@^B	BNN^AA@A
	@^BANANA	ANA@^BAN	
	A@^BANAN	A@^BANAN	
	NA@^BANA	BANANA@^	
	ANA@^BAN	NANA@^BA	
	NANA@^BA	NA@^BANA	
	ANANA@^B	^BANANA@	
	BANANA@^	@^BANANA	

- Reversible

```
function inverseBWT (string s)
  create empty table
  repeat length(s) times
    insert s as a column of table before first column of the table // first insert creates first column
    sort rows of the table alphabetically
  return (row that ends with the 'EOF' character)
```

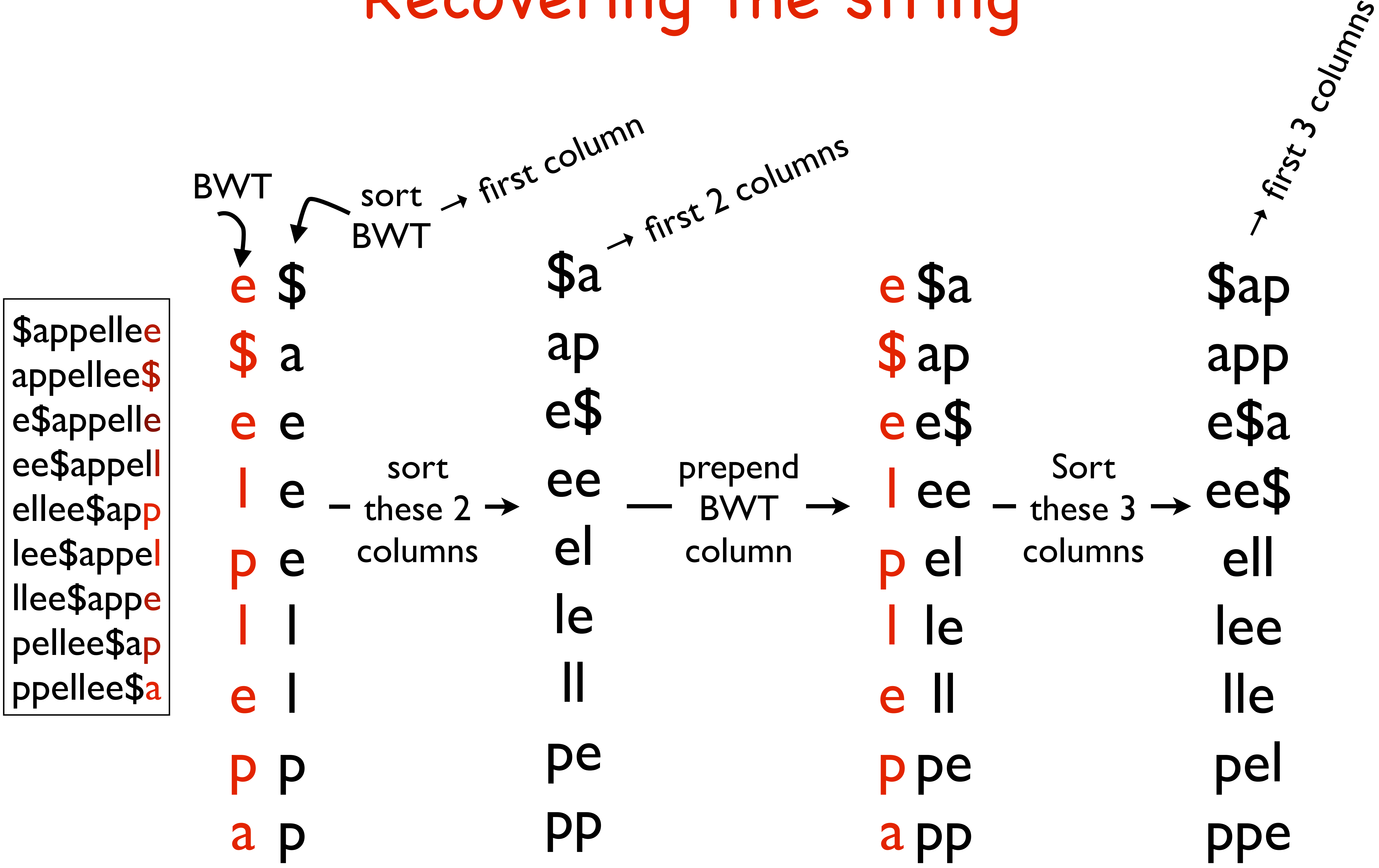
Last column only suffices to reconstruct **entire matrix**, and thus recover **original string**



Add 1	Sort 1	Add 2	Sort 2	Add 3	Sort 3	Add 4	Sort 4
B	A	BA	AN	BAN	ANA	BANA	ANAN
N	A	NA	AN	NAN	ANA	NANA	ANA@
N	A	NA	A@	NA@	A@^	NA@^	A@^B
^	B	^B	BA	^BA	BAN	^BAN	BANA
A	N	AN	NA	ANA	NAN	ANAN	NANA
A	N	AN	NA	ANA	NA@	ANA@	NA@^
@	^	@^	^B	@^B	^BA	@^BA	^BAN
A	@	A@	@^	A@^	@^B	A@^B	@^BA
last	1st col	pairs	2nd col	triples	3rd col	4mers	4thcol

Add 5	Sort 5	Add 6	Sort 6	Add 7	Sort 7	Add 8	Sort 8
BANAN	ANANA	BANANA	ANANA@	BANANA@	ANANA@^	BANANA@^	ANANA@^B
NANA@	ANA@^	NANA@^	ANA@^B	NANA@^B	ANA@^BA	NANA@^BA	ANA@^BAN
NA@^B	A@^BA	NA@^BA	A@^BAN	NA@^BAN	A@^BANA	NA@^BANA	A@^BANAN
^BANA	BANAN	^BANAN	BANANA	^BANANA	BANANA@	^BANANA@	BANANA@^
ANANA	NANA@	ANANA@	NANA@^	ANANA@^	NANA@^B	ANANA@^B	NANA@^BA
ANA@^	NA@^B	ANA@^B	NA@^BA	ANA@^BA	NA@^BAN	ANA@^BAN	NA@^BANA
@^BAN	^BANA	@^BANA	^BANAN	@^BANAN	^BANANA	@^BANANA	^BANANA@
A@^BA	@^BAN	A@^BAN	@^BANA	A@^BANA	@^BANAN	A@^BANAN	@^BANANA
5mers	5thcol	6-mers	6thcol	7-mers	7thcol	8-mers	Full matrix

Recovering the string



$appellee
appellee$
e$appelle
ee$appell
ellee$app
lee$appel
llee$appe
pellee$ap
ppellee$a

BWT

sort BWT → first column

sort these 2 columns → first 2 columns

prepend BWT column

Sort these 3 columns → first 3 columns

e	$	$a	e	$a	$ap
$	a	ap	$	ap	app
e	e	e$	e	e$	e$a
l	e	ee	l	ee	ee$
p	e	el	p	el	ell
l	l	le	l	le	lee
e	l	ll	e	ll	lle
p	p	pe	p	pe	pel
a	p	pp	a	pp	ppe

Inverse BWT

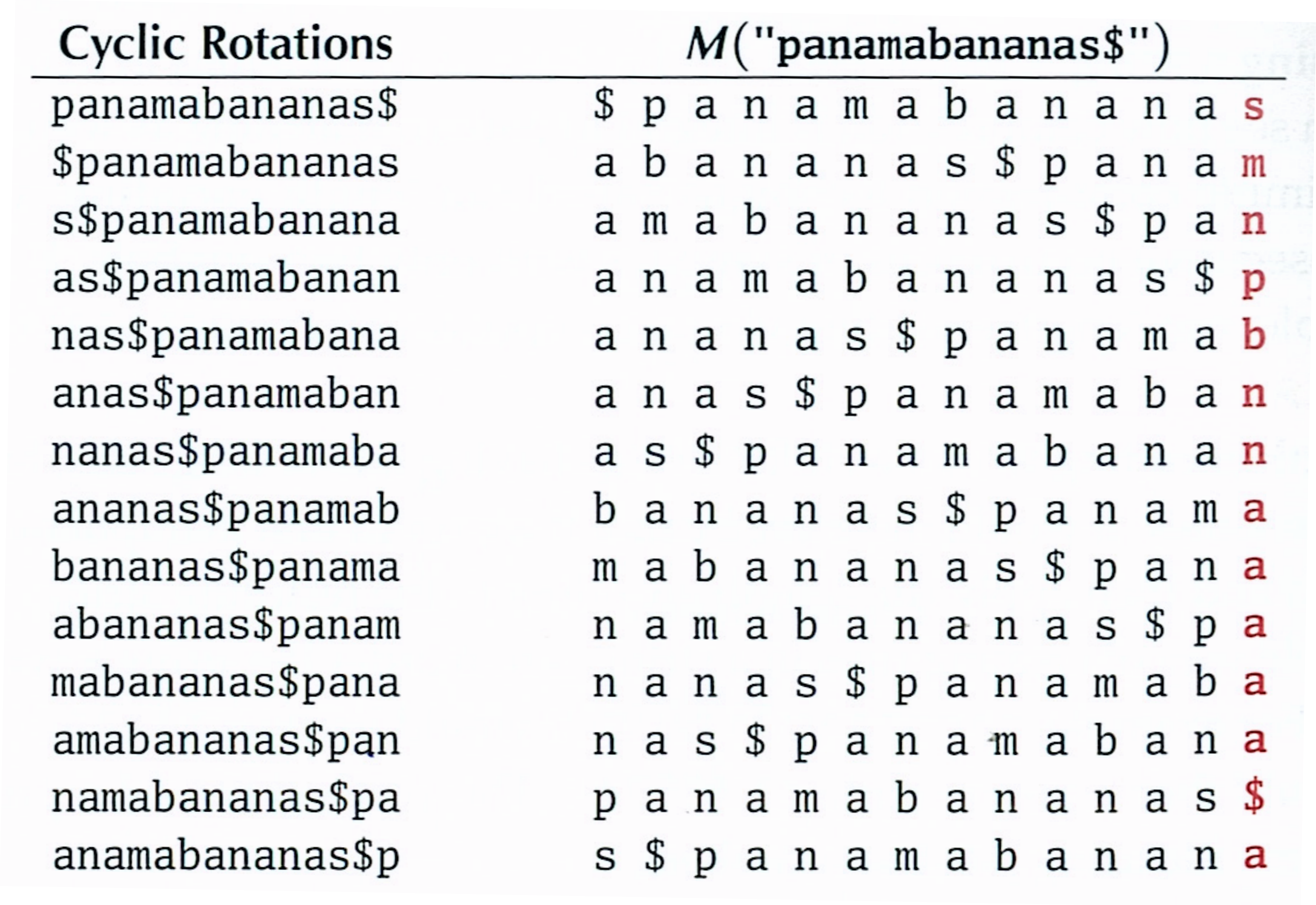
```
def inverseBWT(s):
    B = [$s_1$,$s_2$,$s_3$,...,$s_n$]
    for i = 1..n:
        sort B
        prepend $s_i$ to B[i]
    return row of B that ends with $
```

do$oodwg Another BWT Example

d $	$d	d $d	$do	d$do	$dog	d$dog	$dogw
o d	d$	o d$	d$d	od$d	d$do	od$do	d$dog
$ d	do	$ do	dog	$dog	dogw	$dogw	dogwo
o g	gw	o gw	gwo	ogwo	gwoo	ogwoo	gwood
o o	od	o od	od$	ood$	od$d	ood$d	od$do
d o	og	d og	ogw	dogw	ogwo	dogwo	ogwoo
w o	oo	w oo	ood	wood	ood$	wood$	ood$d
g w	wo	g wo	woo	gwoo	wood	gwood	wood$
Prepend	Sort	Prepend	Sort	Prepend	Sort	Prepend	Sort

d $dogw	$dogwo	d $dogwo	$dogwoo	d $dogwoo	$dogwood
o d$dog	d$dogw	o d$dogw	d$dogwo	o d$dogwo	d$dogwoo
$ dogwo	dogwoo	$ dogwoo	dogwood	$ dogwood	dogwood$
o gwood	gwood$	o gwood$	gwood$d	o gwood$d	gwood$do
o od$do	od$dog	o od$dog	od$dogw	o od$dogw	od$dogwo
d ogwoo	ogwood	d ogwood	ogwood$	d ogwood$	ogwood$d
w ood$d	ood$do	w ood$do	ood$dog	w ood$dog	ood$dogw
g wood$	wood$d	g wood$d	wood$do	g wood$do	wood$dog
Prepend	Sort	Prepend	Sort	Prepend	Sort

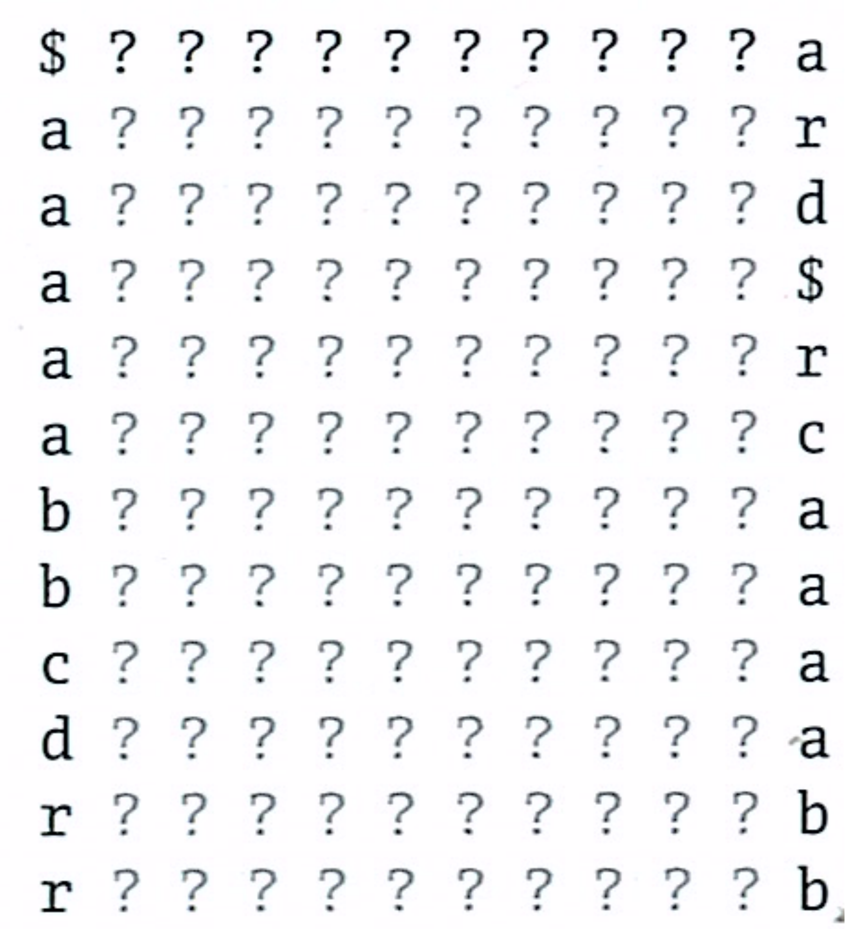
All cyclic rotation of string *panamabananas$*, note that the first row of matrix M contains the word with the special character in first position



Cyclic Rotations	*M*("panamabananas$")
panamabananas$	$ p a n a m a b a n a n a s
$panamabananas	a b a n a n a s $ p a n a m
s$panamabanana	a m a b a n a n a s $ p a n
as$panamabanan	a n a m a b a n a n a s $ p
nas$panamabana	a n a n a s $ p a n a m a b
anas$panamaban	a n a s $ p a n a m a b a n
nanas$panamaba	a s $ p a n a m a b a n a n
ananas$panamab	b a n a n a s $ p a n a m a
bananas$panama	m a b a n a n a s $ p a n a
abananas$panam	n a m a b a n a n a s $ p a
mabananas$pana	n a n a s $ p a n a m a b a
amabananas$pan	n a s $ p a n a m a b a n a
namabananas$pa	p a n a m a b a n a n a s $
anamabananas$p	s $ p a n a m a b a n a n a

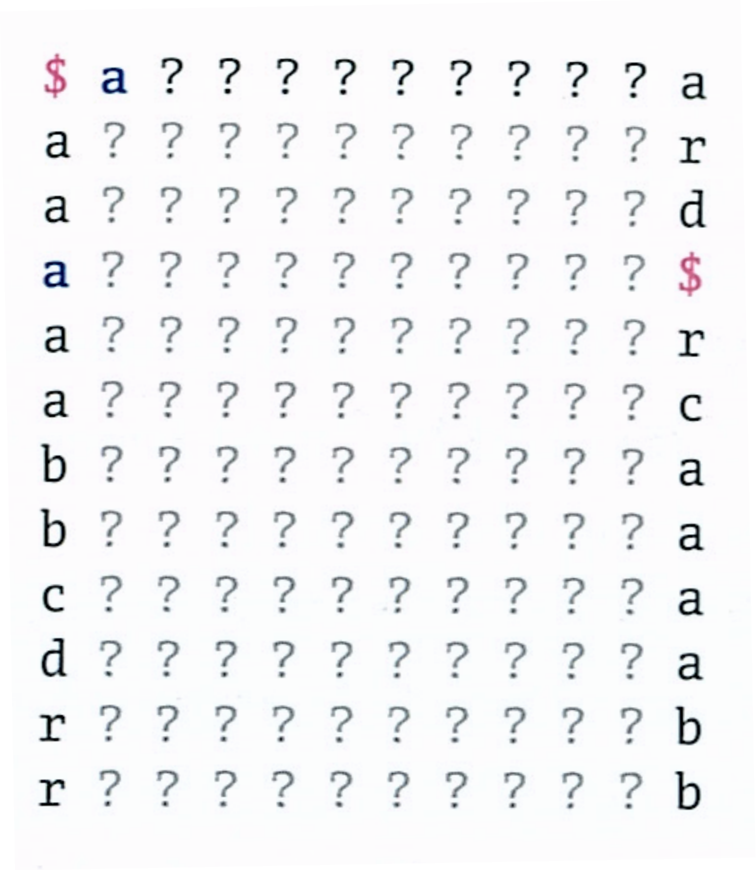
If we determine the first row of M(*Text*) then we can move the $ to the end of this row to reproduce *Text.*

How do we determine the remaining symbols on this first row, if all we know is *FirstColumn* and *LastColumn*?



$	?	?	?	?	?	?	?	?	?	?	a
a	?	?	?	?	?	?	?	?	?	?	r
a	?	?	?	?	?	?	?	?	?	?	d
a	?	?	?	?	?	?	?	?	?	?	$
a	?	?	?	?	?	?	?	?	?	?	r
a	?	?	?	?	?	?	?	?	?	?	c
b	?	?	?	?	?	?	?	?	?	?	a
b	?	?	?	?	?	?	?	?	?	?	a
c	?	?	?	?	?	?	?	?	?	?	a
d	?	?	?	?	?	?	?	?	?	?	a
r	?	?	?	?	?	?	?	?	?	?	b
r	?	?	?	?	?	?	?	?	?	?	b

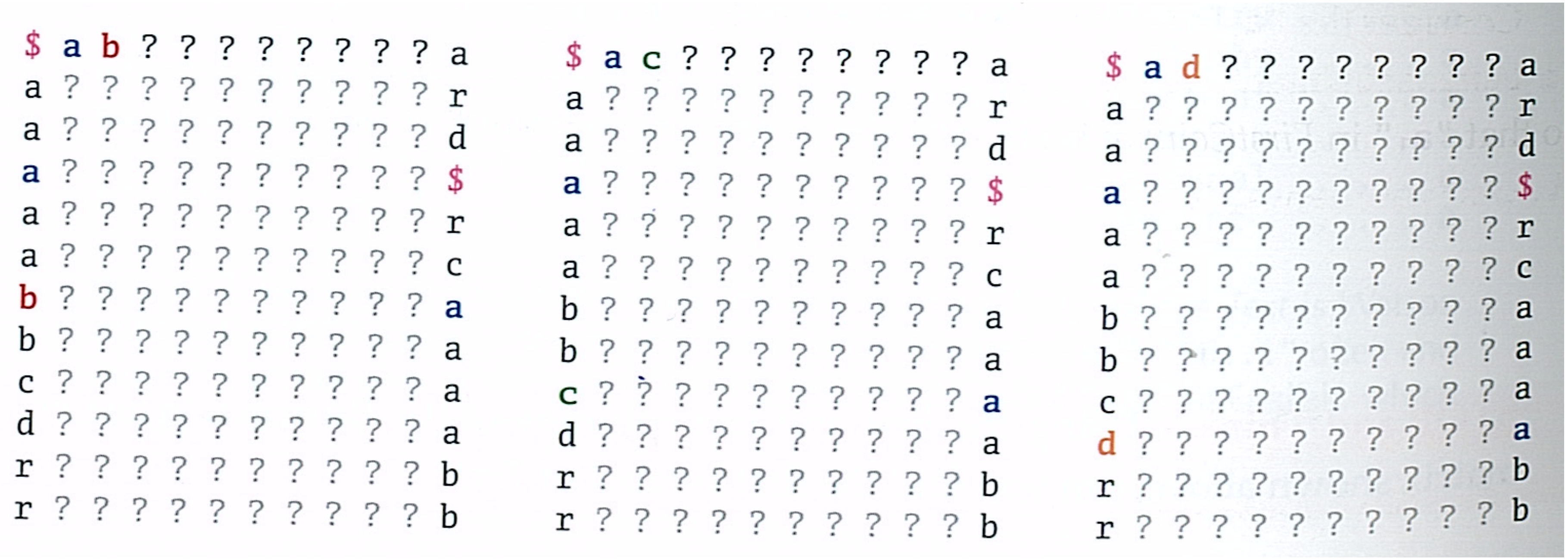
Note that the first symbol in **Text** must follow *$* in any cyclic rotation of **Text**.



$	a	?	?	?	?	?	?	?	?	?	a
a	?	?	?	?	?	?	?	?	?	?	r
a	?	?	?	?	?	?	?	?	?	?	d
a	?	?	?	?	?	?	?	?	?	?	$
a	?	?	?	?	?	?	?	?	?	?	r
a	?	?	?	?	?	?	?	?	?	?	c
b	?	?	?	?	?	?	?	?	?	?	a
b	?	?	?	?	?	?	?	?	?	?	a
c	?	?	?	?	?	?	?	?	?	?	a
d	?	?	?	?	?	?	?	?	?	?	a
r	?	?	?	?	?	?	?	?	?	?	b
r	?	?	?	?	?	?	?	?	?	?	b

Rule1: The symbol in **LastColumn** must precede the symbol of **Text** found in the same row of the **FirstColumn**

The next symbol of Text should be the first symbol in a row of M(**Text)** that end in *a*. But, five rows and in an *a* and we don't known which of them is the correct one!



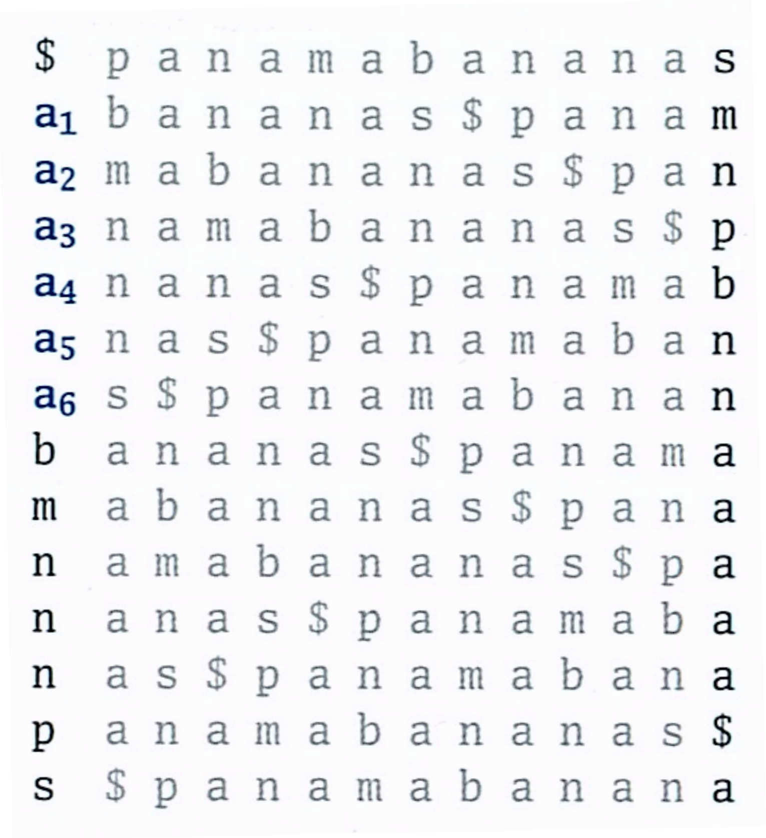
$	a	b	?	?	?	?	?	?	?	?	a
a	?	?	?	?	?	?	?	?	?	?	r
a	?	?	?	?	?	?	?	?	?	?	d
a	?	?	?	?	?	?	?	?	?	?	$
a	?	?	?	?	?	?	?	?	?	?	r
a	?	?	?	?	?	?	?	?	?	?	c
b	?	?	?	?	?	?	?	?	?	?	a
b	?	?	?	?	?	?	?	?	?	?	a
c	?	?	?	?	?	?	?	?	?	?	a
d	?	?	?	?	?	?	?	?	?	?	a
r	?	?	?	?	?	?	?	?	?	?	b
r	?	?	?	?	?	?	?	?	?	?	b

$	a	c	?	?	?	?	?	?	?	?	a
a	?	?	?	?	?	?	?	?	?	?	r
a	?	?	?	?	?	?	?	?	?	?	d
a	?	?	?	?	?	?	?	?	?	?	$
a	?	?	?	?	?	?	?	?	?	?	r
a	?	?	?	?	?	?	?	?	?	?	c
b	?	?	?	?	?	?	?	?	?	?	a
b	?	?	?	?	?	?	?	?	?	?	a
c	?	?	?	?	?	?	?	?	?	?	a
d	?	?	?	?	?	?	?	?	?	?	a
r	?	?	?	?	?	?	?	?	?	?	b
r	?	?	?	?	?	?	?	?	?	?	b

$	a	d	?	?	?	?	?	?	?	?	a
a	?	?	?	?	?	?	?	?	?	?	r
a	?	?	?	?	?	?	?	?	?	?	d
a	?	?	?	?	?	?	?	?	?	?	$
a	?	?	?	?	?	?	?	?	?	?	r
a	?	?	?	?	?	?	?	?	?	?	c
b	?	?	?	?	?	?	?	?	?	?	a
b	?	?	?	?	?	?	?	?	?	?	a
c	?	?	?	?	?	?	?	?	?	?	a
d	?	?	?	?	?	?	?	?	?	?	a
r	?	?	?	?	?	?	?	?	?	?	b
r	?	?	?	?	?	?	?	?	?	?	b

The three possibilities for the third element of the first row of M(Text) when BWT(Text) is *ard$rcaaaabb*. How would you choose among *b*, *c* and *d* for the second symbol of **Text**?

To determine the remaining symbols of *Text* , we need to define the **First-Last Property**. We can rank the six instances of a appear in *FirstColumn* and we say that they have ranks from 1 to 6.



```
$  p a n a m a b a n a n a s
a1 b a n a n a s $ p a n a m
a2 m a b a n a n a s $ p a n
a3 n a m a b a n a n a s $ p
a4 n a n a s $ p a n a m a b
a5 n a s $ p a n a m a b a n
a6 s $ p a n a m a b a n a n
b  a n a n a s $ p a n a m a
m  a b a n a n a s $ p a n a
n  a m a b a n a n a s $ p a
n  a n a s $ p a n a m a b a
n  a s $ p a n a m a b a n a
p  a n a m a b a n a n a s $
s  $ p a n a m a b a n a n a
```

Considering $\mathtt{a1}$ in the FirstColumn

$\mathtt{a_1bananas\$panam}$

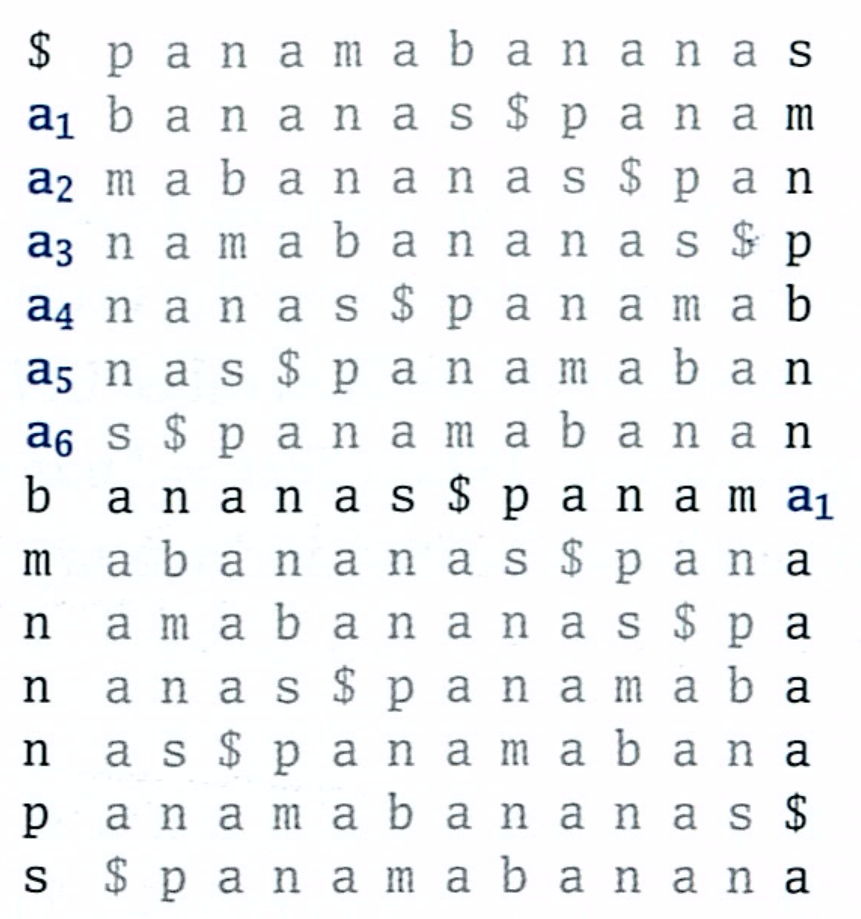
If we cyclically rotate this string, we obtain:

$\mathtt{panama_1bananas\$}$

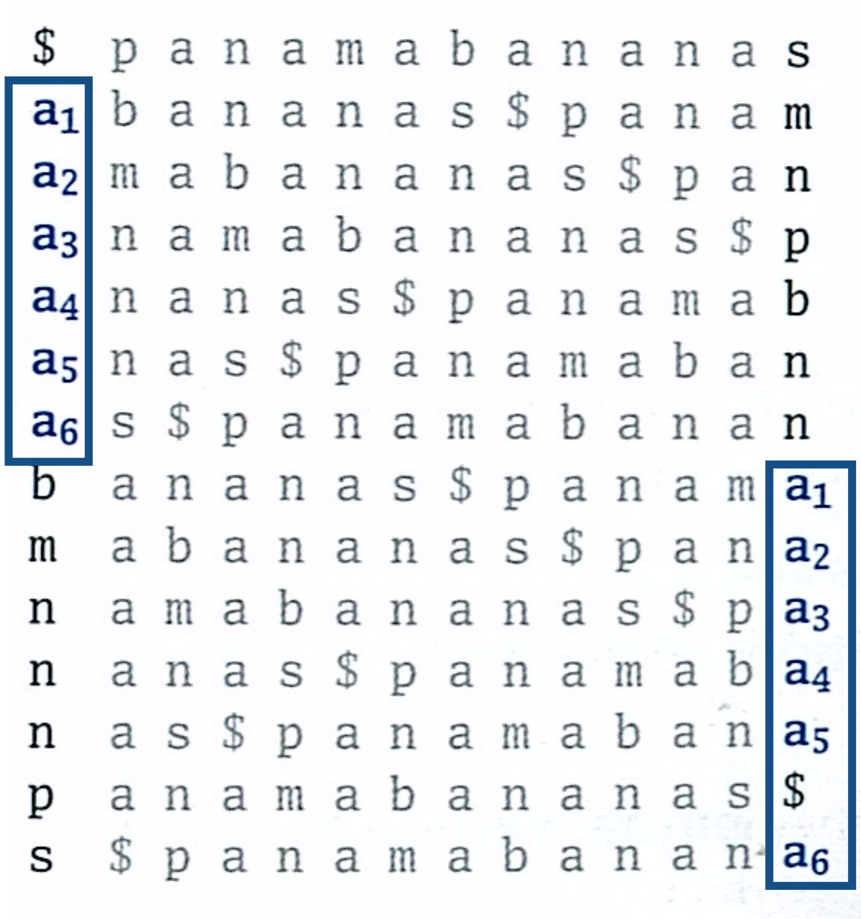
Now we can identify the positions of the other five instances:

$\mathtt{pa_3na_2ma_1ba_4na_5na_6s\$}$

Where are the other five instance of a located in LastColumn?

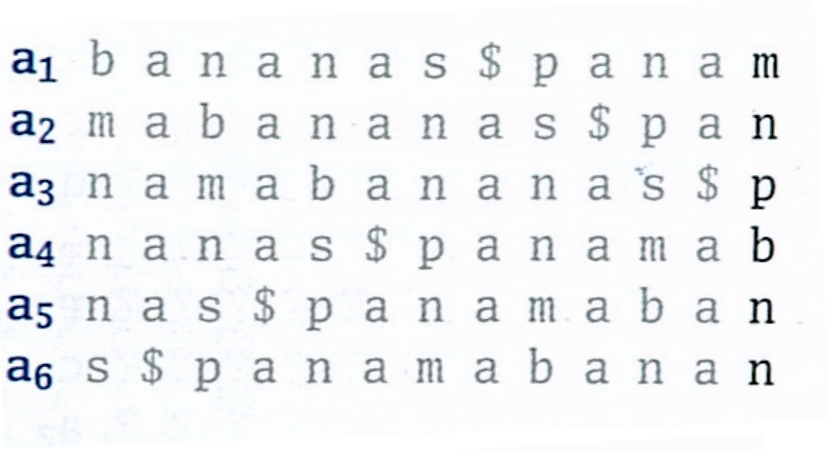


```
$   p a n a m a b a n a n a s
a_1 b a n a n a s $ p a n a m
a_2 m a b a n a n a s $ p a n
a_3 n a m a b a n a n a s $ p
a_4 n a n a s $ p a n a m a b
a_5 n a s $ p a n a m a b a n
a_6 s $ p a n a m a b a n a n
b   a n a n a s $ p a n a m a_1
m   a b a n a n a s $ p a n a
n   a m a b a n a n a s $ p a
n   a n a s $ p a n a m a b a
n   a s $ p a n a m a b a n a
p   a n a m a b a n a n a s $
s   $ p a n a m a b a n a n a
```



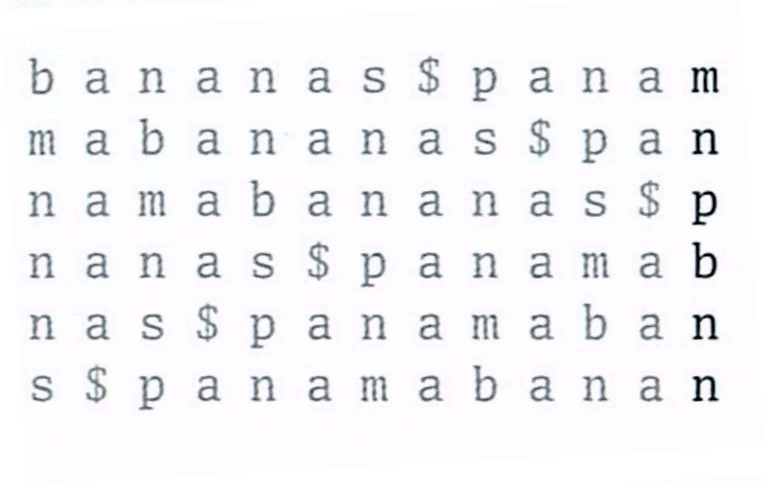
```
$   p a n a m a b a n a n a s
a_1 b a n a n a s $ p a n a m
a_2 m a b a n a n a s $ p a n
a_3 n a m a b a n a n a s $ p
a_4 n a n a s $ p a n a m a b
a_5 n a s $ p a n a m a b a n
a_6 s $ p a n a m a b a n a n
b   a n a n a s $ p a n a m a_1
m   a b a n a n a s $ p a n a_2
n   a m a b a n a n a s $ p a_3
n   a n a s $ p a n a m a b a_4
n   a s $ p a n a m a b a n a_5
p   a n a m a b a n a n a s $
s   $ p a n a m a b a n a n a_6
```

Rule2: The *k-th* occurrence of symbol in **FirstColumn** and the *k-th* occurrence of symbol in **LastColumn** correspond to the same position of symbol in *Text*.



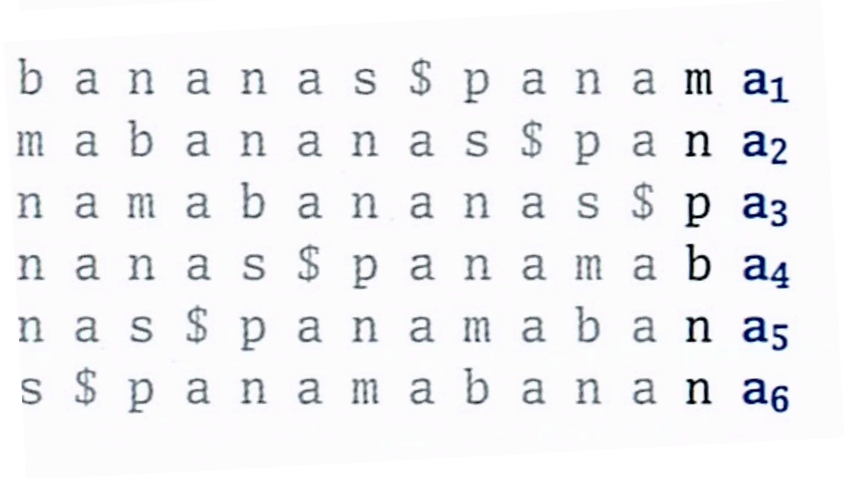
```
a1 b a n a n a s $ p a n a m
a2 m a b a n a n a s $ p a n
a3 n a m a b a n a n a s $ p
a4 n a n a s $ p a n a m a b
a5 n a s $ p a n a m a b a n
a6 s $ p a n a m a b a n a n
```

The rows are already ordered in lexicographically, so if we chop off the a from the beginning of each row the remaining strings should still be ordered lexicographically:



```
b a n a n a s $ p a n a m
m a b a n a n a s $ p a n
n a m a b a n a n a s $ p
n a n a s $ p a n a m a b
n a s $ p a n a m a b a n
s $ p a n a m a b a n a n
```

Adding a back to the end of each row should not change the lexicographic ordering of these rows:

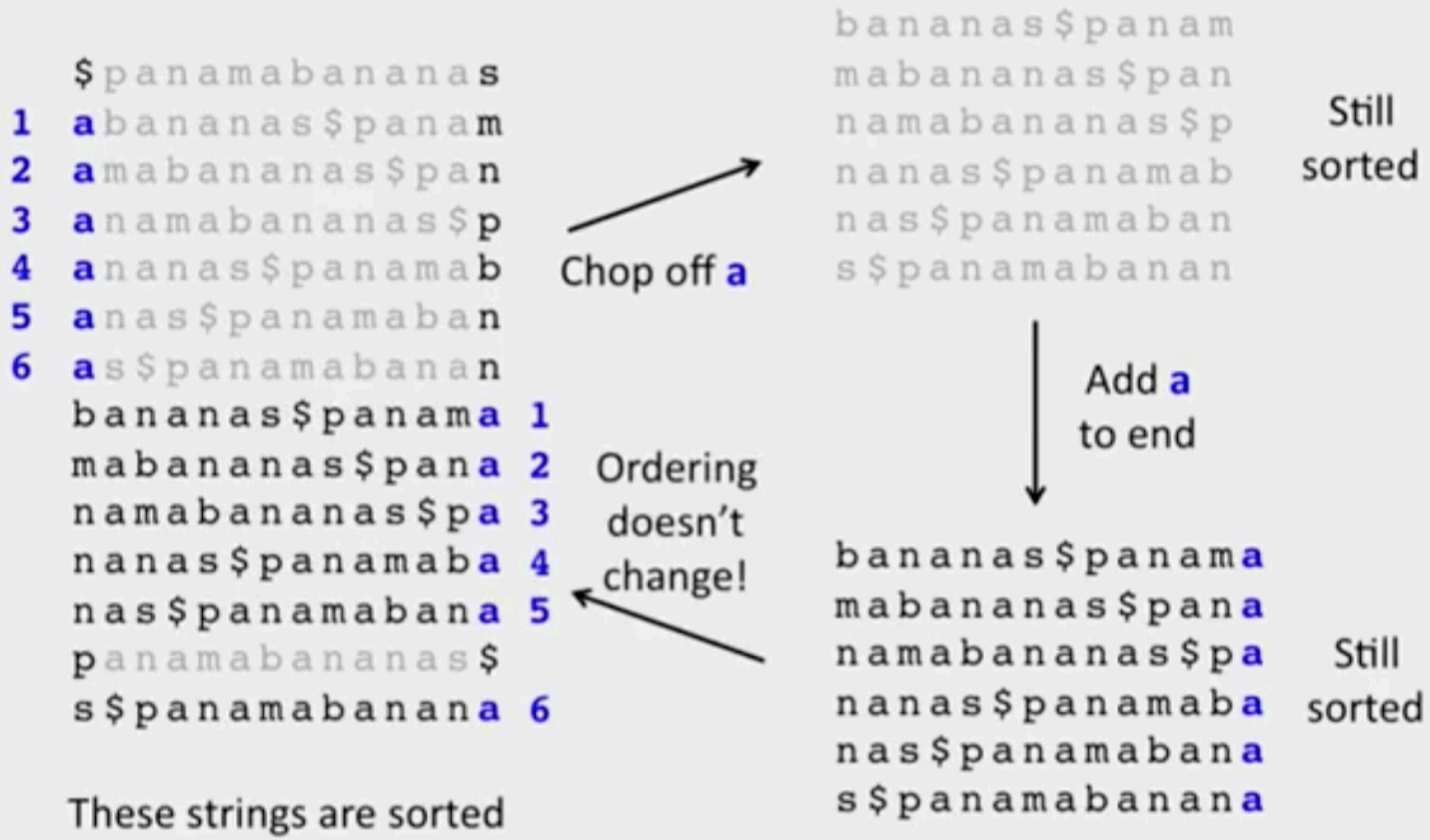


```
b a n a n a s $ p a n a m a1
m a b a n a n a s $ p a n a2
n a m a b a n a n a s $ p a3
n a n a s $ p a n a m a b a4
n a s $ p a n a m a b a n a5
s $ p a n a m a b a n a n a6
```

$pa_3na_2ma_1ba_4na_5na_6s\$$

The **Rule2** can be of course generalised for any possible symbol and any string Text.

Is It True in General?



```
  $panamabananas
1 abananas$panam
2 amabananas$pan
3 anamabananas$p
4 ananas$panamab
5 anas$panamaban
6 as$panamabanan
  bananas$panama 1
  mabananas$pana 2
  namabananas$pa 3
  nanas$panamaba 4
  nas$panamabana 5
  panamabananas$
  s$panamabanana 6
```

These strings are sorted

Chop off a

```
bananas$panam
mabananas$pan
namabananas$p
nanas$panamab
nas$panamaban
s$panamabanan
```

Still sorted

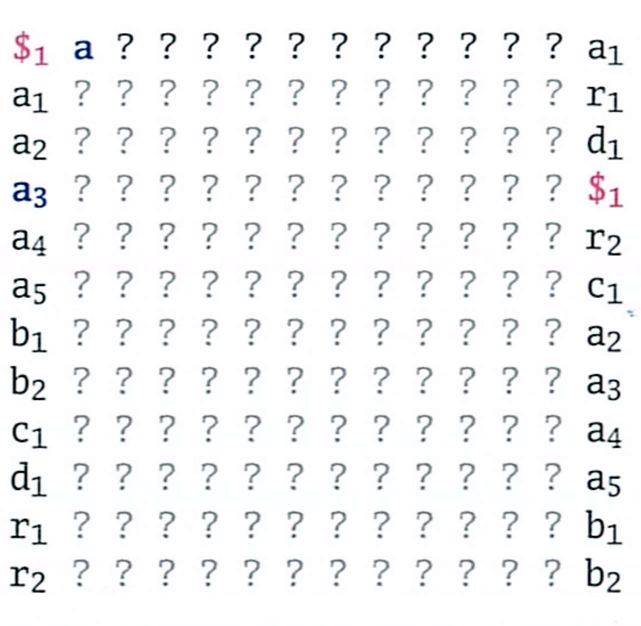
Add a to end

```
bananas$panama
mabananas$pana
namabananas$pa
nanas$panamaba
nas$panamabana
s$panamabanana
```

Still sorted

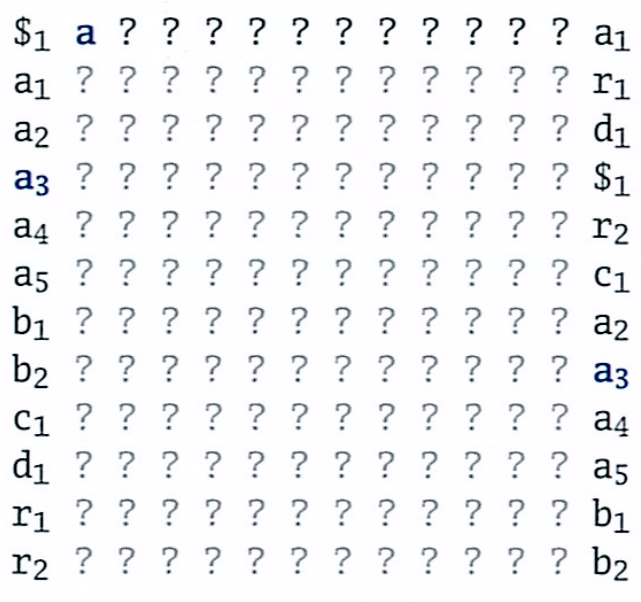
Ordering doesn't change!

How do we can use the **FirstLast Property** for the **Burrows-Wheeler inversion**



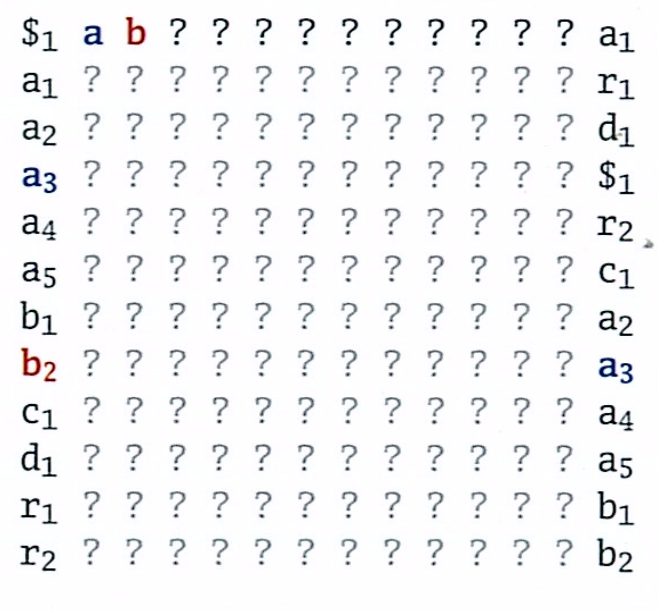
$\$_1$	a	?	?	?	?	?	?	?	?	?	?	?	a_1
a_1	?	?	?	?	?	?	?	?	?	?	?	?	r_1
a_2	?	?	?	?	?	?	?	?	?	?	?	?	d_1
a_3	?	?	?	?	?	?	?	?	?	?	?	?	$\$_1$
a_4	?	?	?	?	?	?	?	?	?	?	?	?	r_2
a_5	?	?	?	?	?	?	?	?	?	?	?	?	c_1
b_1	?	?	?	?	?	?	?	?	?	?	?	?	a_2
b_2	?	?	?	?	?	?	?	?	?	?	?	?	a_3
c_1	?	?	?	?	?	?	?	?	?	?	?	?	a_4
d_1	?	?	?	?	?	?	?	?	?	?	?	?	a_5
r_1	?	?	?	?	?	?	?	?	?	?	?	?	b_1
r_2	?	?	?	?	?	?	?	?	?	?	?	?	b_2

The **FL prop** reveals where **a3** is hiding in *LastColumn*:



$\$_1$	a	?	?	?	?	?	?	?	?	?	?	?	a_1
a_1	?	?	?	?	?	?	?	?	?	?	?	?	r_1
a_2	?	?	?	?	?	?	?	?	?	?	?	?	d_1
a_3	?	?	?	?	?	?	?	?	?	?	?	?	$\$_1$
a_4	?	?	?	?	?	?	?	?	?	?	?	?	r_2
a_5	?	?	?	?	?	?	?	?	?	?	?	?	c_1
b_1	?	?	?	?	?	?	?	?	?	?	?	?	a_2
b_2	?	?	?	?	?	?	?	?	?	?	?	?	a_3
c_1	?	?	?	?	?	?	?	?	?	?	?	?	a_4
d_1	?	?	?	?	?	?	?	?	?	?	?	?	a_5
r_1	?	?	?	?	?	?	?	?	?	?	?	?	b_1
r_2	?	?	?	?	?	?	?	?	?	?	?	?	b_2

Since we know that **a3** is located ate the end of the eighth row, we can wrap around this row to determine that **b2** follows **a3** in the *Text*

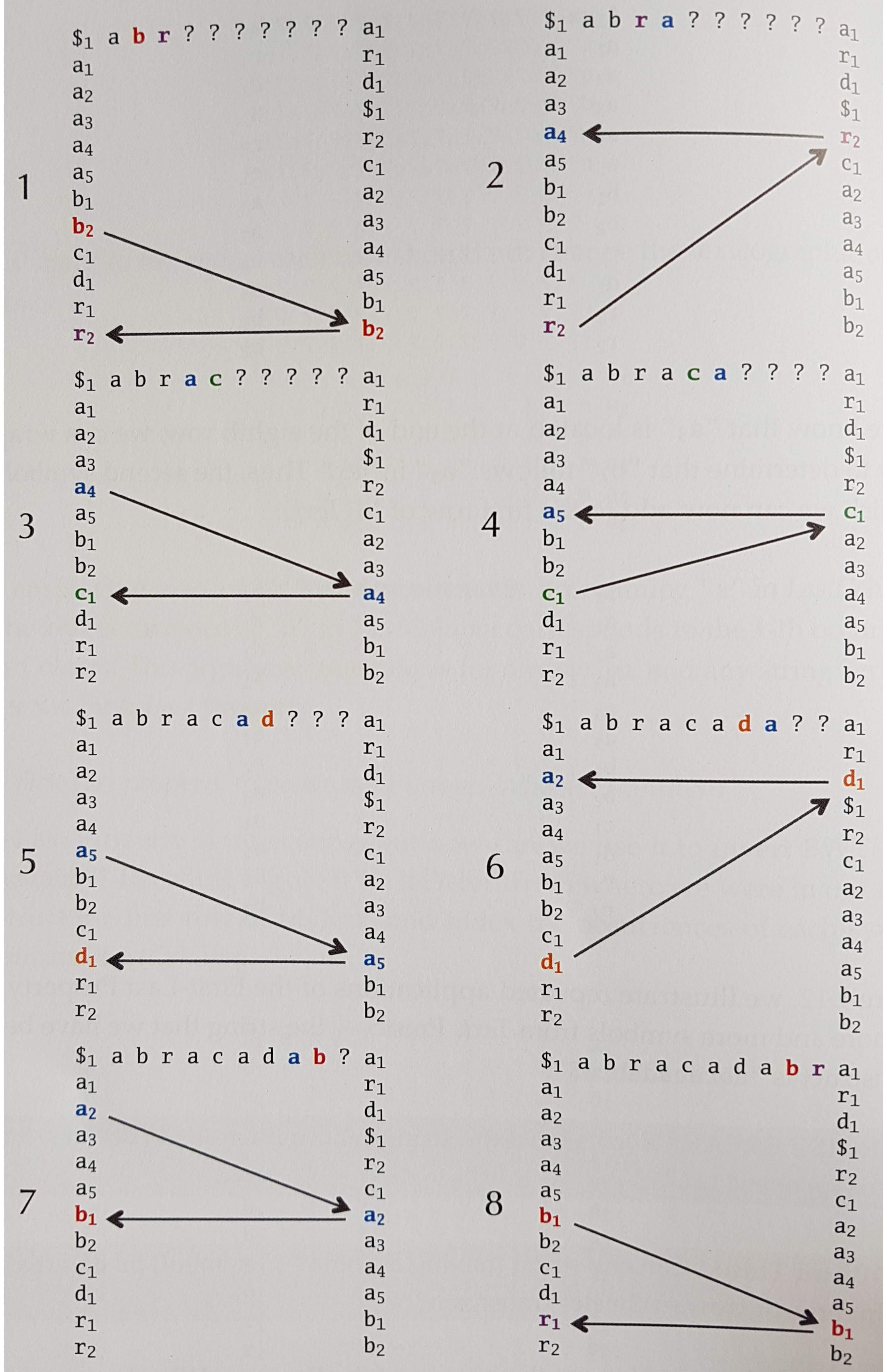


$\$_1$	a	b	?	?	?	?	?	?	?	?	?	?	a_1
a_1	?	?	?	?	?	?	?	?	?	?	?	?	r_1
a_2	?	?	?	?	?	?	?	?	?	?	?	?	d_1
a_3	?	?	?	?	?	?	?	?	?	?	?	?	$\$_1$
a_4	?	?	?	?	?	?	?	?	?	?	?	?	r_2
a_5	?	?	?	?	?	?	?	?	?	?	?	?	c_1
b_1	?	?	?	?	?	?	?	?	?	?	?	?	a_2
b_2	?	?	?	?	?	?	?	?	?	?	?	?	a_3
c_1	?	?	?	?	?	?	?	?	?	?	?	?	a_4
d_1	?	?	?	?	?	?	?	?	?	?	?	?	a_5
r_1	?	?	?	?	?	?	?	?	?	?	?	?	b_1
r_2	?	?	?	?	?	?	?	?	?	?	?	?	b_2

Application of **Rule 1**

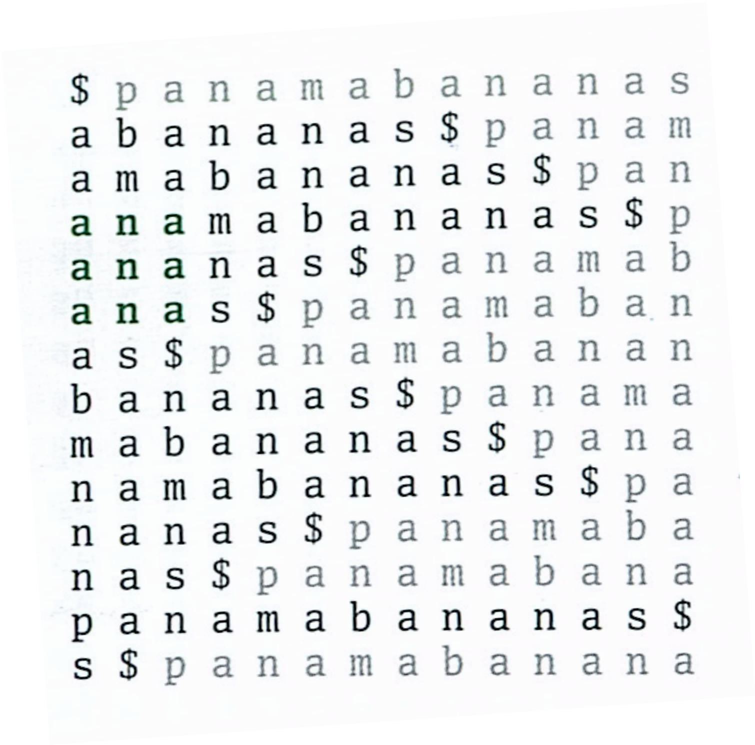
Inverting the BWT

Repeated applications of the **FL prop** to reconstruct the *Text* from its **BWT**


1
$1 a b r ? ? ? ? ? ? ? a1
2
$1 a b r a ? ? ? ? ? ? a1
3
$1 a b r a c ? ? ? ? ? a1
4
$1 a b r a c a ? ? ? ? a1
5
$1 a b r a c a d ? ? ? a1
6
$1 a b r a c a d a ? ? a1
7
$1 a b r a c a d a b ? a1
8
$1 a b r a c a d a b r a1
a1 a2 a3 a4 a5 b1 b2 c1 d1 r1 r2
r1 d1 $1 r2 c1 a2 a3 a4 a5 b1 b2

Pattern matching with BWT

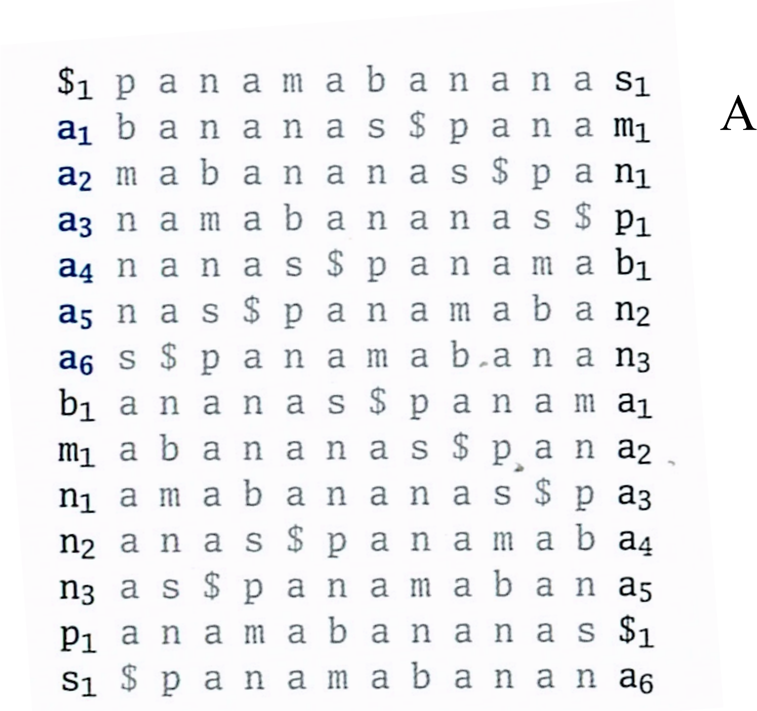
Each row of M(*Text*) begins with a different suffix of the *Text*. Since these suffixes are already ordered lexicographically, any matches of *Pattern* in *Text* will clump together at the beginning of rows of M(*Text*).



```
$ p a n a m a b a n a n a s
a b a n a n a s $ p a n a m
a m a b a n a n a s $ p a n
a n a m a b a n a n a s $ p
a n a n a s $ p a n a m a b
a n a s $ p a n a m a b a n
a s $ p a n a m a b a n a n
b a n a n a s $ p a n a m a
m a b a n a n a s $ p a n a
n a m a b a n a n a s $ p a
n a n a s $ p a n a m a b a
n a s $ p a n a m a b a n a
p a n a m a b a n a n a s $
s $ p a n a m a b a n a n a
```

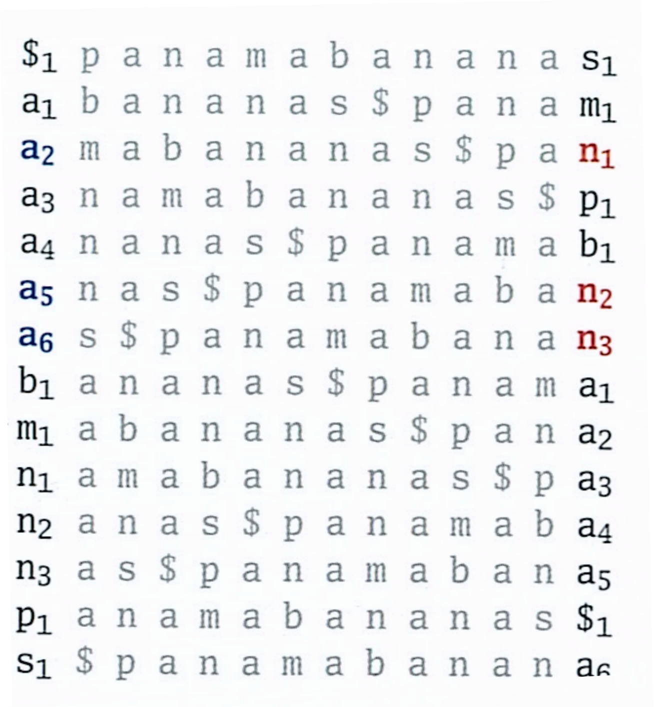
How do we do pattern matching without knowing the entire matrix **M(Text)?**

String to search **ana**



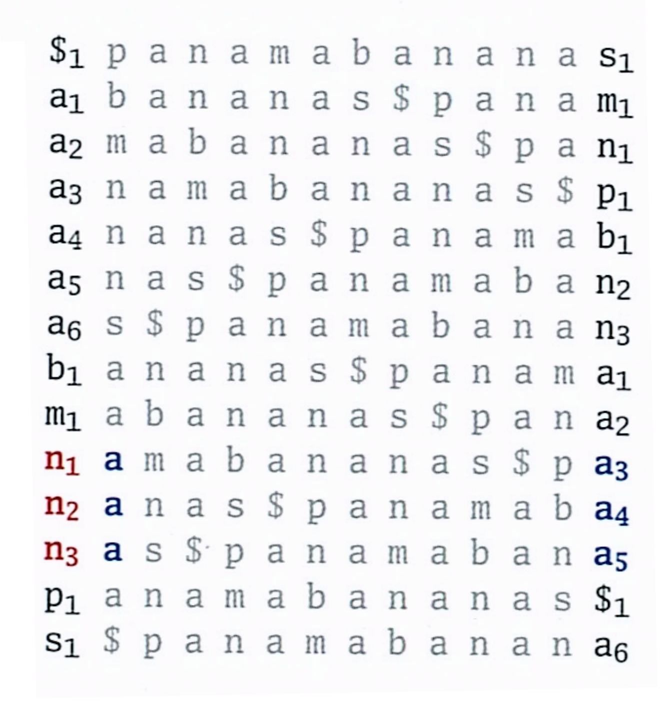
$\$_1$	p	a	n	a	m	a	b	a	n	a	n	a	s_1
a_1	b	a	n	a	n	a	s	$	p	a	n	a	m_1
a_2	m	a	b	a	n	a	n	a	s	$	p	a	n_1
a_3	n	a	m	a	b	a	n	a	n	a	s	$	p_1
a_4	n	a	n	a	s	$	p	a	n	a	m	a	b_1
a_5	n	a	s	$	p	a	n	a	m	a	b	a	n_2
a_6	s	$	p	a	n	a	m	a	b	a	n	a	n_3
b_1	a	n	a	n	a	s	$	p	a	n	a	m	a_1
m_1	a	b	a	n	a	n	a	s	$	p	a	n	a_2
n_1	a	m	a	b	a	n	a	n	a	s	$	p	a_3
n_2	a	n	a	s	$	p	a	n	a	m	a	b	a_4
n_3	a	s	$	p	a	n	a	m	a	b	a	n	a_5
p_1	a	n	a	m	a	b	a	n	a	n	a	s	$\$_1$
s_1	$	p	a	n	a	m	a	b	a	n	a	n	a_6

Application of **Rule 2**



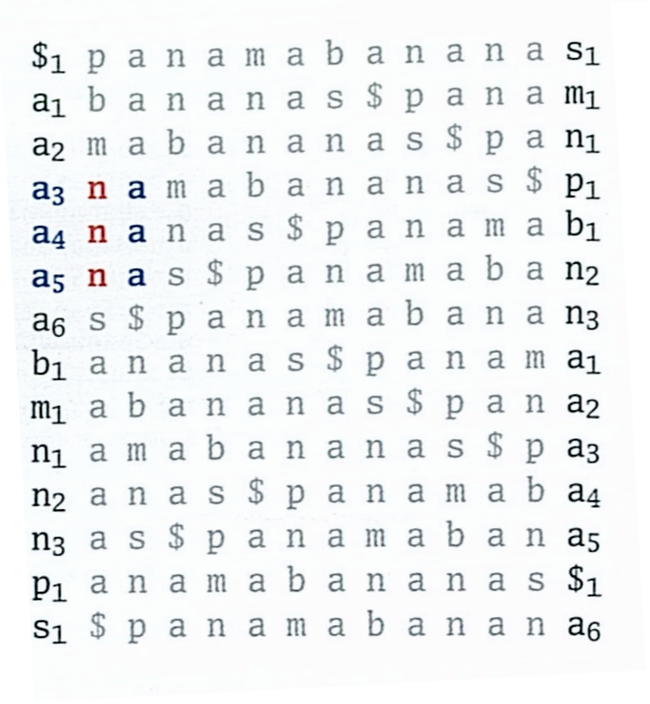
$\$_1$	p	a	n	a	m	a	b	a	n	a	n	a	s_1
a_1	b	a	n	a	n	a	s	$	p	a	n	a	m_1
a_2	m	a	b	a	n	a	n	a	s	$	p	a	n_1
a_3	n	a	m	a	b	a	n	a	n	a	s	$	p_1
a_4	n	a	n	a	s	$	p	a	n	a	m	a	b_1
a_5	n	a	s	$	p	a	n	a	m	a	b	a	n_2
a_6	s	$	p	a	n	a	m	a	b	a	n	a	n_3
b_1	a	n	a	n	a	s	$	p	a	n	a	m	a_1
m_1	a	b	a	n	a	n	a	s	$	p	a	n	a_2
n_1	a	m	a	b	a	n	a	n	a	s	$	p	a_3
n_2	a	n	a	s	$	p	a	n	a	m	a	b	a_4
n_3	a	s	$	p	a	n	a	m	a	b	a	n	a_5
p_1	a	n	a	m	a	b	a	n	a	n	a	s	$\$_1$
s_1	$	p	a	n	a	m	a	b	a	n	a	n	a_6

Three **a** are associated to an **n** in the **LastColumn**



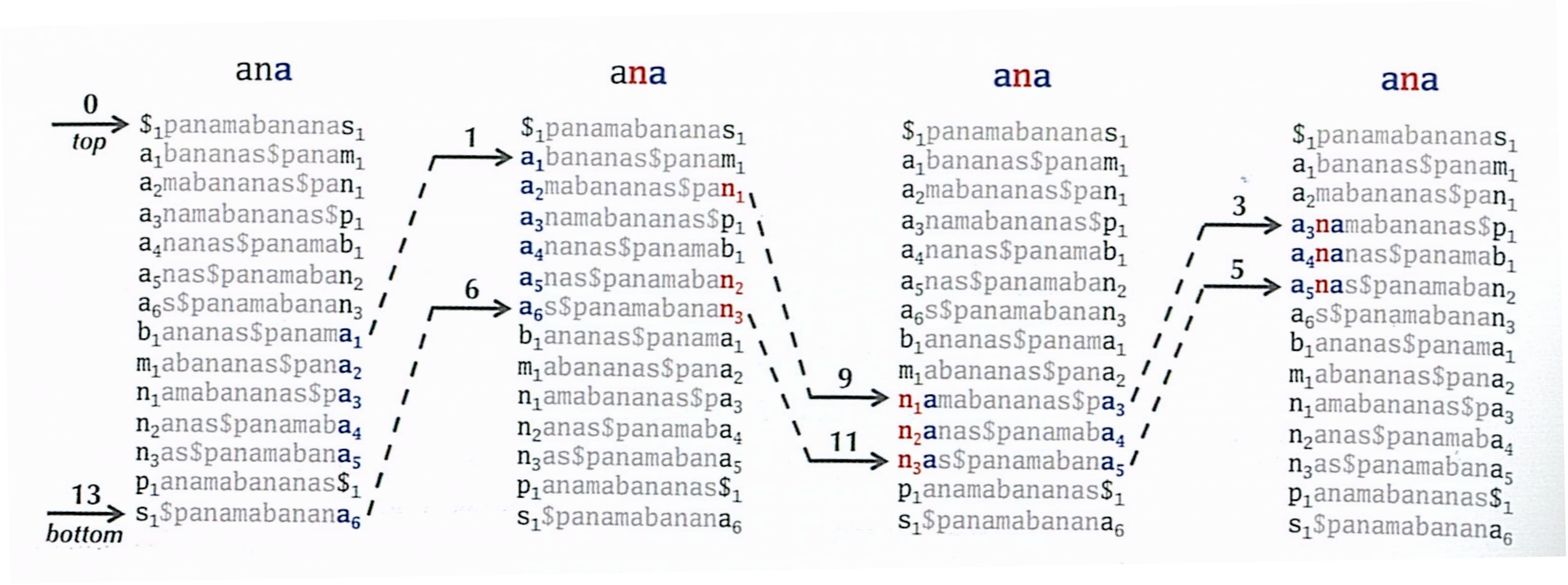
$\$_1$	p	a	n	a	m	a	b	a	n	a	n	a	s_1
a_1	b	a	n	a	n	a	s	$	p	a	n	a	m_1
a_2	m	a	b	a	n	a	n	a	s	$	p	a	n_1
a_3	n	a	m	a	b	a	n	a	n	a	s	$	p_1
a_4	n	a	n	a	s	$	p	a	n	a	m	a	b_1
a_5	n	a	s	$	p	a	n	a	m	a	b	a	n_2
a_6	s	$	p	a	n	a	m	a	b	a	n	a	n_3
b_1	a	n	a	n	a	s	$	p	a	n	a	m	a_1
m_1	a	b	a	n	a	n	a	s	$	p	a	n	a_2
n_1	a	m	a	b	a	n	a	n	a	s	$	p	a_3
n_2	a	n	a	s	$	p	a	n	a	m	a	b	a_4
n_3	a	s	$	p	a	n	a	m	a	b	a	n	a_5
p_1	a	n	a	m	a	b	a	n	a	n	a	s	$\$_1$
s_1	$	p	a	n	a	m	a	b	a	n	a	n	a_6

All three **n** are associated to an **a** in the **FirstColumn**

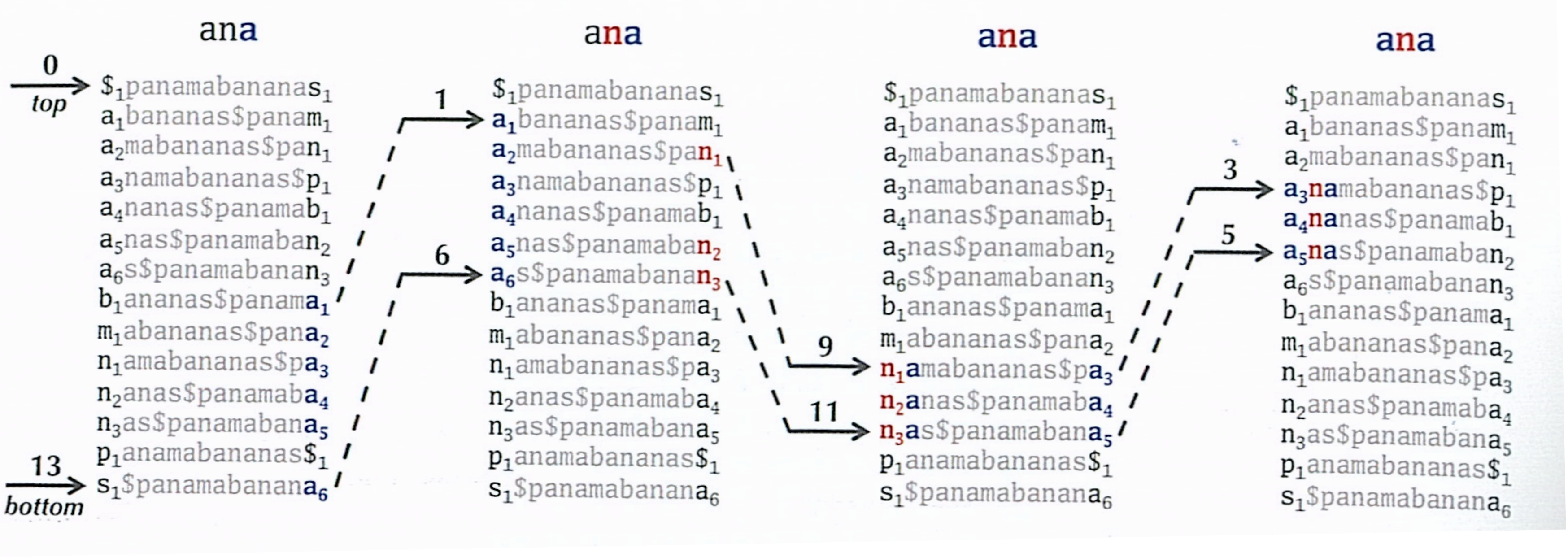


$\$_1$	p	a	n	a	m	a	b	a	n	a	n	a	s_1
a_1	b	a	n	a	n	a	s	$	p	a	n	a	m_1
a_2	m	a	b	a	n	a	n	a	s	$	p	a	n_1
a_3	n	a	m	a	b	a	n	a	n	a	s	$	p_1
a_4	n	a	n	a	s	$	p	a	n	a	m	a	b_1
a_5	n	a	s	$	p	a	n	a	m	a	b	a	n_2
a_6	s	$	p	a	n	a	m	a	b	a	n	a	n_3
b_1	a	n	a	n	a	s	$	p	a	n	a	m	a_1
m_1	a	b	a	n	a	n	a	s	$	p	a	n	a_2
n_1	a	m	a	b	a	n	a	n	a	s	$	p	a_3
n_2	a	n	a	s	$	p	a	n	a	m	a	b	a_4
n_3	a	s	$	p	a	n	a	m	a	b	a	n	a_5
p_1	a	n	a	m	a	b	a	n	a	n	a	s	$\$_1$
s_1	$	p	a	n	a	m	a	b	a	n	a	n	a_6

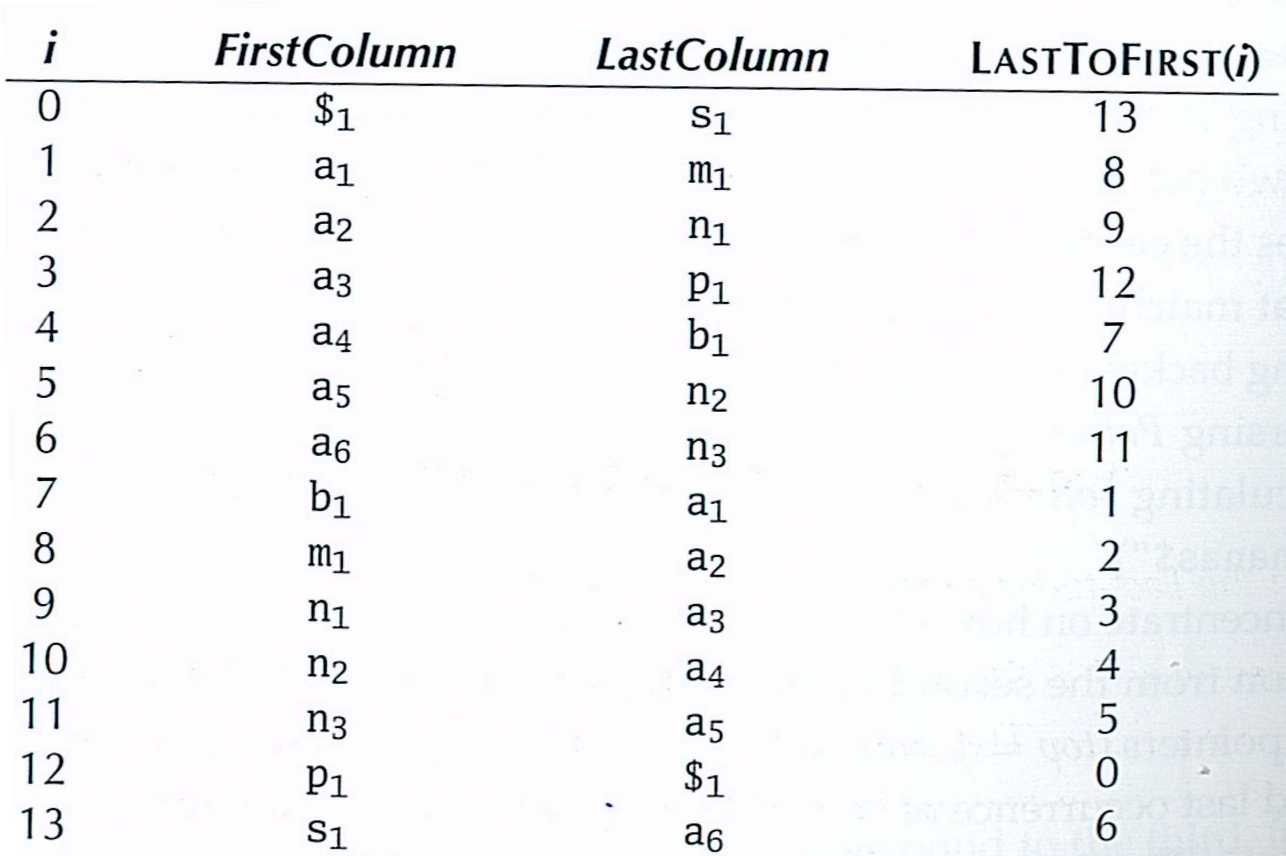
Pattern matching with BWT - Using pointers


ana
0
top
$\mathtt{\$_1panamabananas_1}$
$\mathtt{a_1bananas\$panam_1}$
$\mathtt{a_2mabananas\$pan_1}$
$\mathtt{a_3namabananas\$p_1}$
$\mathtt{a_4nanas\$panamab_1}$
$\mathtt{a_5nas\$panamaban_2}$
$\mathtt{a_6s\$panamabanan_3}$
$\mathtt{b_1ananas\$panama_1}$
$\mathtt{m_1abananas\$pana_2}$
$\mathtt{n_1amabananas\$pa_3}$
$\mathtt{n_2anas\$panamaba_4}$
$\mathtt{n_3as\$panamabana_5}$
$\mathtt{p_1anamabananas\$_1}$
$\mathtt{s_1\$panamabanana_6}$
13
bottom
1
6
ana
$\mathtt{\$_1panamabananas_1}$
$\mathtt{a_1bananas\$panam_1}$
$\mathtt{a_2mabananas\$pan_1}$
$\mathtt{a_3namabananas\$p_1}$
$\mathtt{a_4nanas\$panamab_1}$
$\mathtt{a_5nas\$panamaban_2}$
$\mathtt{a_6s\$panamabanan_3}$
$\mathtt{b_1ananas\$panama_1}$
$\mathtt{m_1abananas\$pana_2}$
$\mathtt{n_1amabananas\$pa_3}$
$\mathtt{n_2anas\$panamaba_4}$
$\mathtt{n_3as\$panamabana_5}$
$\mathtt{p_1anamabananas\$_1}$
$\mathtt{s_1\$panamabanana_6}$
9
11
ana
$\mathtt{\$_1panamabananas_1}$
$\mathtt{a_1bananas\$panam_1}$
$\mathtt{a_2mabananas\$pan_1}$
$\mathtt{a_3namabananas\$p_1}$
$\mathtt{a_4nanas\$panamab_1}$
$\mathtt{a_5nas\$panamaban_2}$
$\mathtt{a_6s\$panamabanan_3}$
$\mathtt{b_1ananas\$panama_1}$
$\mathtt{m_1abananas\$pana_2}$
$\mathtt{n_1amabananas\$pa_3}$
$\mathtt{n_2anas\$panamaba_4}$
$\mathtt{n_3as\$panamabana_5}$
$\mathtt{p_1anamabananas\$_1}$
$\mathtt{s_1\$panamabanana_6}$
3
5
ana
$\mathtt{\$_1panamabananas_1}$
$\mathtt{a_1bananas\$panam_1}$
$\mathtt{a_2mabananas\$pan_1}$
$\mathtt{a_3namabananas\$p_1}$
$\mathtt{a_4nanas\$panamab_1}$
$\mathtt{a_5nas\$panamaban_2}$
$\mathtt{a_6s\$panamabanan_3}$
$\mathtt{b_1ananas\$panama_1}$
$\mathtt{m_1abananas\$pana_2}$
$\mathtt{n_1amabananas\$pa_3}$
$\mathtt{n_2anas\$panamaba_4}$
$\mathtt{n_3as\$panamabana_5}$
$\mathtt{p_1anamabananas\$_1}$
$\mathtt{s_1\$panamabanana_6}$

The pointers top and bottom hold the indices of the first and last rows of M(*Text*) matching the current suffix of *Pattern*. The above diagram shows how these pointers are updated when walking backwards through **`ana`** and looking for substring matches in **`panamabananas$`**

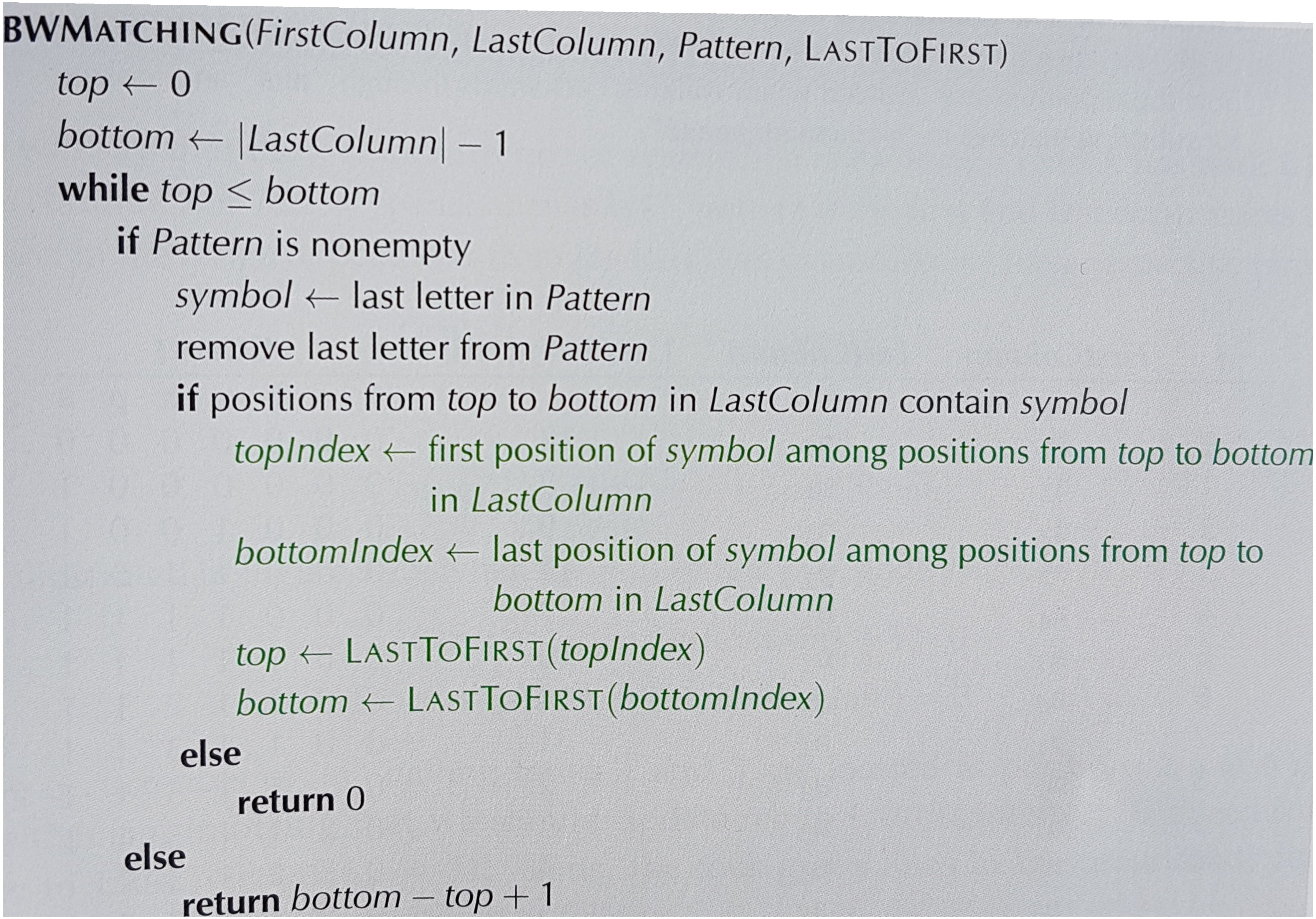

ana
0
top
$\mathrm{\$_1panamabananas_1}$
$\mathrm{a_1bananas\$panam_1}$
$\mathrm{a_2mabananas\$pan_1}$
$\mathrm{a_3namabananas\$p_1}$
$\mathrm{a_4nanas\$panamab_1}$
$\mathrm{a_5nas\$panamaban_2}$
$\mathrm{a_6s\$panamabanan_3}$
$\mathrm{b_1ananas\$panama_1}$
$\mathrm{m_1abananas\$pana_2}$
$\mathrm{n_1amabananas\$pa_3}$
$\mathrm{n_2anas\$panamaba_4}$
$\mathrm{n_3as\$panamabana_5}$
$\mathrm{p_1anamabananas\$_1}$
$\mathrm{s_1\$panamabanana_6}$
13
bottom
ana
1
6
$\mathrm{\$_1panamabananas_1}$
$\mathrm{a_1bananas\$panam_1}$
$\mathrm{a_2mabananas\$pan_1}$
$\mathrm{a_3namabananas\$p_1}$
$\mathrm{a_4nanas\$panamab_1}$
$\mathrm{a_5nas\$panamaban_2}$
$\mathrm{a_6s\$panamabanan_3}$
$\mathrm{b_1ananas\$panama_1}$
$\mathrm{m_1abananas\$pana_2}$
$\mathrm{n_1amabananas\$pa_3}$
$\mathrm{n_2anas\$panamaba_4}$
$\mathrm{n_3as\$panamabana_5}$
$\mathrm{p_1anamabananas\$_1}$
$\mathrm{s_1\$panamabanana_6}$
ana
9
11
$\mathrm{\$_1panamabananas_1}$
$\mathrm{a_1bananas\$panam_1}$
$\mathrm{a_2mabananas\$pan_1}$
$\mathrm{a_3namabananas\$p_1}$
$\mathrm{a_4nanas\$panamab_1}$
$\mathrm{a_5nas\$panamaban_2}$
$\mathrm{a_6s\$panamabanan_3}$
$\mathrm{b_1ananas\$panama_1}$
$\mathrm{m_1abananas\$pana_2}$
$\mathrm{n_1amabananas\$pa_3}$
$\mathrm{n_2anas\$panamaba_4}$
$\mathrm{n_3as\$panamabana_5}$
$\mathrm{p_1anamabananas\$_1}$
$\mathrm{s_1\$panamabanana_6}$
ana
3
5
$\mathrm{\$_1panamabananas_1}$
$\mathrm{a_1bananas\$panam_1}$
$\mathrm{a_2mabananas\$pan_1}$
$\mathrm{a_3namabananas\$p_1}$
$\mathrm{a_4nanas\$panamab_1}$
$\mathrm{a_5nas\$panamaban_2}$
$\mathrm{a_6s\$panamabanan_3}$
$\mathrm{b_1ananas\$panama_1}$
$\mathrm{m_1abananas\$pana_2}$
$\mathrm{n_1amabananas\$pa_3}$
$\mathrm{n_2anas\$panamaba_4}$
$\mathrm{n_3as\$panamabana_5}$
$\mathrm{p_1anamabananas\$_1}$
$\mathrm{s_1\$panamabanana_6}$

Given a symbol at position i of **LastColumn**, the *Last-to-First* mapping identifies this symbol's position in **FirstColumn**

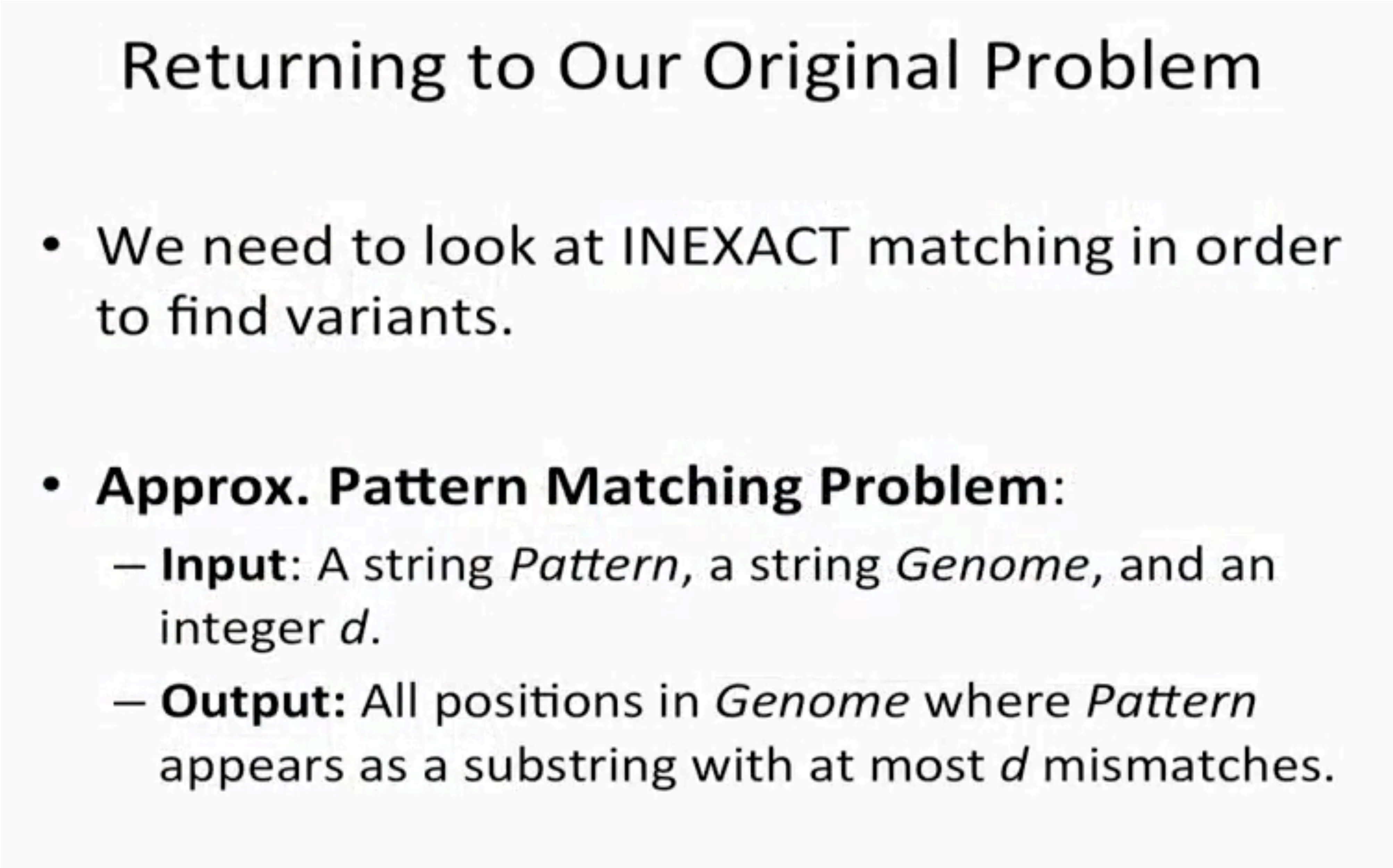


i	*FirstColumn*	*LastColumn*	LastToFirst(*i*)
0	$\$_1$	s_1	13
1	a_1	m_1	8
2	a_2	n_1	9
3	a_3	p_1	12
4	a_4	b_1	7
5	a_5	n_2	10
6	a_6	n_3	11
7	b_1	a_1	1
8	m_1	a_2	2
9	n_1	a_3	3
10	n_2	a_4	4
11	n_3	a_5	5
12	p_1	$\$_1$	0
13	s_1	a_6	6

Pattern matching with BWT



```
BWMATCHING(FirstColumn, LastColumn, Pattern, LASTTOFIRST)
    top ← 0
    bottom ← |LastColumn| − 1
    while top ≤ bottom
        if Pattern is nonempty
            symbol ← last letter in Pattern
            remove last letter from Pattern
            if positions from top to bottom in LastColumn contain symbol
                topIndex ← first position of symbol among positions from top to bottom
                                in LastColumn
                bottomIndex ← last position of symbol among positions from top to
                                bottom in LastColumn
                top ← LASTTOFIRST(topIndex)
                bottom ← LASTTOFIRST(bottomIndex)
            else
                return 0
        else
            return bottom − top + 1
```



Returning to Our Original Problem

- We need to look at INEXACT matching in order to find variants.

- **Approx. Pattern Matching Problem**:
  - **Input**: A string *Pattern*, a string *Genome*, and an integer *d*.
  - **Output:** All positions in *Genome* where *Pattern* appears as a substring with at most *d* mismatches.

Returning to Our Original Problem

- We need to look at INEXACT matching in order to find variants.

- Multiple **Approx. Pattern Matching Problem**:
  - **Input**: A **collection** of strings *Patterns*, a string *Genome*, and an integer *d*.
  - **Output:** All positions in *Genome* where a string from *Patterns* appears as a substring with at most *d* mismatches.

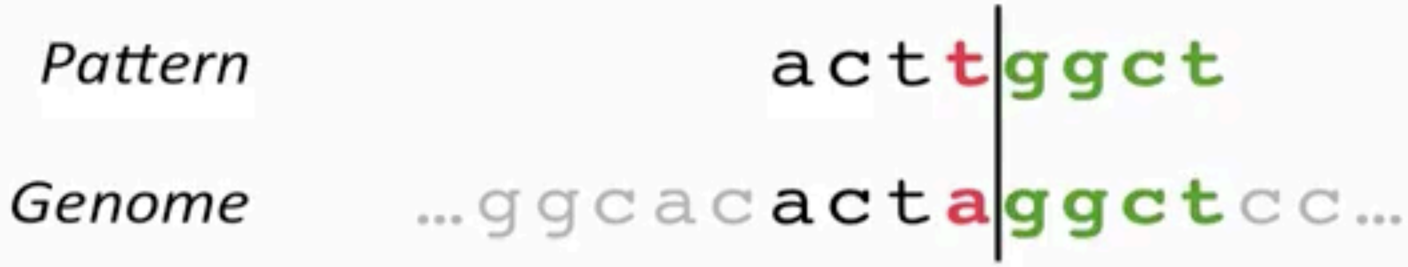
Method 1: Seeding

- Say that *Pattern* appears in *Genome* with 1 mismatch:

```
Pattern              acttggct
Genome   ...ggcacactaggctcc...
```

Method 1: Seeding

- Say that *Pattern* appears in *Genome* with 1 mismatch:



```
Pattern          actt|ggct
Genome   ...ggcacacta|ggctcc...
```

- One of the substrings must match!

Method 1: Seeding

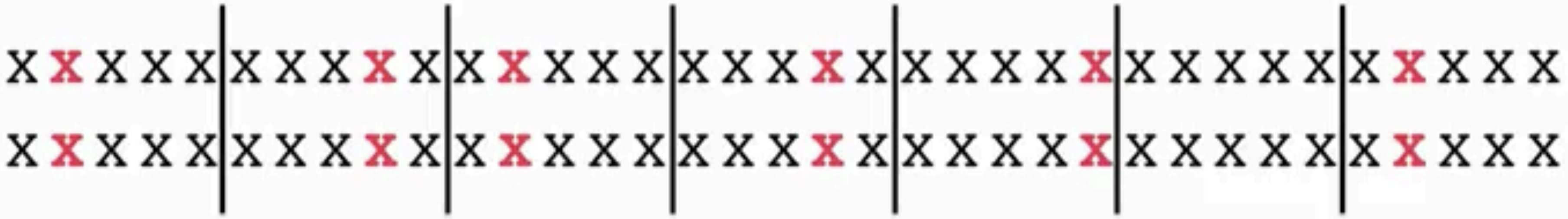
- **Theorem:** If *Pattern* occurs in *Genome* with *d* mismatches, then we can divide *Pattern* into *d* + 1 “equal” pieces and find at least one exact match.

x **x** x x x x x x **x** x x **x** x x x x x x **x** x x x x x **x** x x x x x x **x** x x x

x **x** x x x x x x **x** x x **x** x x x x x x **x** x x x x x **x** x x x x x x **x** x x x

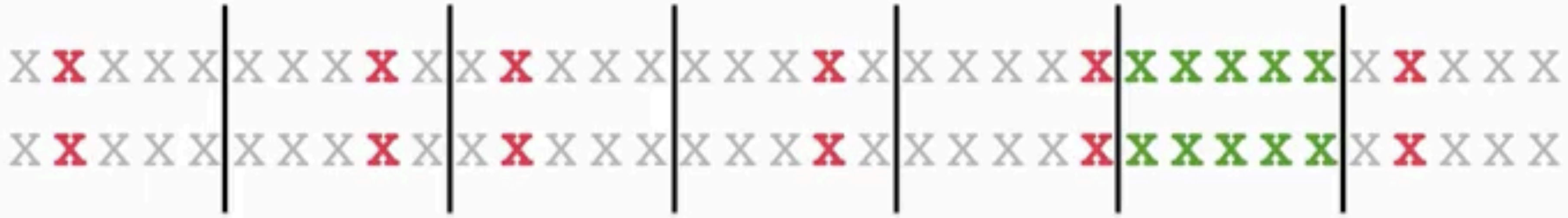
Method 1: Seeding

- **Theorem:** If *Pattern* occurs in *Genome* with *d* mismatches, then we can divide *Pattern* into *d* + 1 “equal” pieces and find at least one exact match.



Method 1: Seeding

- **Theorem:** If *Pattern* occurs in *Genome* with *d* mismatches, then we can divide *Pattern* into *d* + 1 “equal” pieces and find at least one exact match.



Method 1: Seeding

- Say we are looking for at most *d* mismatches.
- Divide each of our strings into *d* + 1 smaller pieces, called **seeds**.

Method 2: BWT Saves the Day Again

- Recall: searching for ana in panamabananas

$\$_{1}\,\texttt{panamabanana}\,s_{1}$
$a_{1}\,\texttt{bananas\$pana}\,m_{1}$
$a_{2}\,\texttt{mabananas\$pa}\,n_{1}$
$a_{3}\,\texttt{namabananas\$}\,p_{1}$
$a_{4}\,\texttt{nanas\$panama}\,b_{1}$
$a_{5}\,\texttt{nas\$panamaba}\,n_{2}$
$a_{6}\,\texttt{s\$panamabana}\,n_{3}$
$b_{1}\,\texttt{ananas\$panam}\,a_{1}$
$m_{1}\,\texttt{abananas\$pan}\,a_{2}$
$n_{1}\,\texttt{amabananas\$p}\,a_{3}$
$n_{2}\,\texttt{anas\$panamab}\,a_{4}$
$n_{3}\,\texttt{as\$panamaban}\,a_{5}$
$p_{1}\,\texttt{anamabananas}\,\$_{1}$
$s_{1}\,\texttt{\$panamabanan}\,a_{6}$

Method 2: BWT Saves the Day Again

- Recall: searching for ana in panamabananas

$\$_1\,\text{panamabanana}\,s_1$

$a_1\,\text{bananas\$pana}\,m_1$

$a_2\,\text{mabananas\$pa}\,n_1$

$a_3\,\text{namabananas\$}\,p_1$

$a_4\,\text{nanas\$panama}\,b_1$

$a_5\,\text{nas\$panamaba}\,n_2$

$a_6\,\text{s\$panamabana}\,n_3$

$b_1\,\text{ananas\$panam}\,a_1$

$m_1\,\text{abananas\$pan}\,a_2$

$n_1\,\text{amabananas\$p}\,a_3$

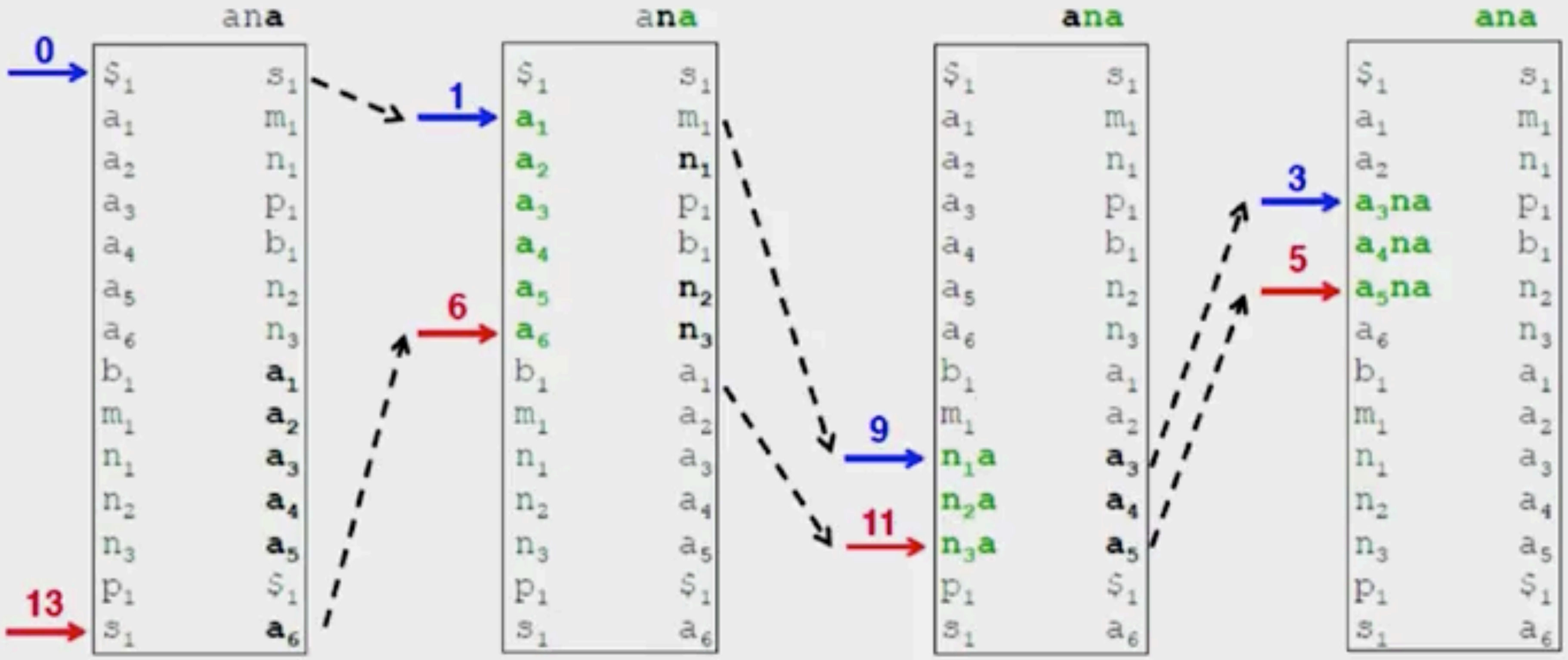
$n_2\,\text{anas\$panamab}\,a_4$

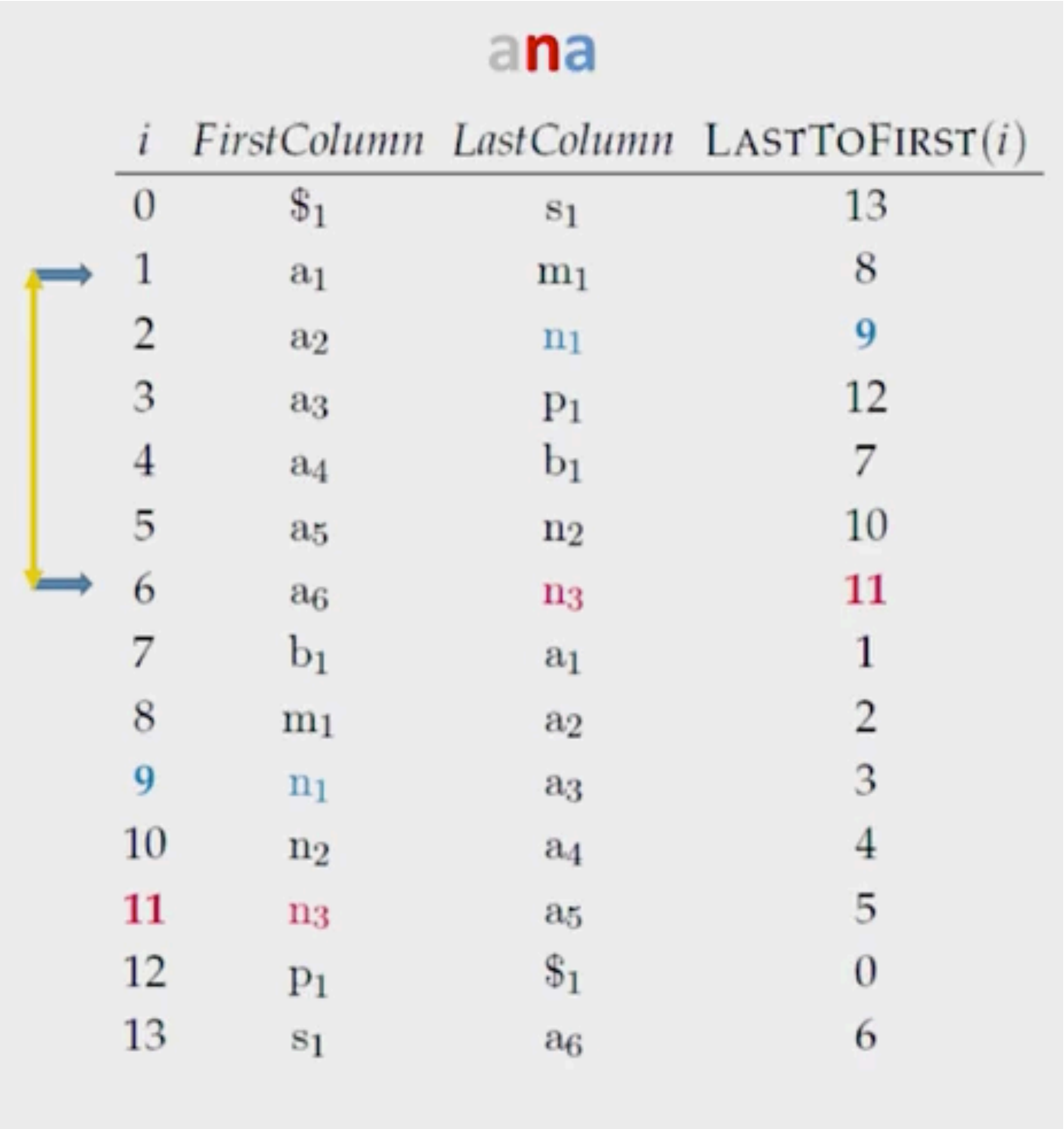
$n_3\,\text{as\$panamaban}\,a_5$

$p_1\,\text{anamabananas}\,\$_1$

$s_1\,\text{\$panamabanan}\,a_6$

Pattern Matching Using BWT


ana
ana
ana
ana
0
13
1
6
9
11
3
5
$\$_1$ a_1 a_2 a_3 a_4 a_5 a_6 b_1 m_1 n_1 n_2 n_3 p_1 s_1
s_1 m_1 n_1 p_1 b_1 n_2 n_3 a_1 a_2 a_3 a_4 a_5 $\$_1$ a_6
n_1a
n_2a
n_3a
a_3na
a_4na
a_5na



ana

i	*FirstColumn*	*LastColumn*	LASTTOFIRST(*i*)
0	$\$_1$	s_1	13
1	a_1	m_1	8
2	a_2	n_1	**9**
3	a_3	p_1	12
4	a_4	b_1	7
5	a_5	n_2	10
6	a_6	n_3	**11**
7	b_1	a_1	1
8	m_1	a_2	2
9	n_1	a_3	3
10	n_2	a_4	4
11	n_3	a_5	5
12	p_1	$\$_1$	0
13	s_1	a_6	6

To complete the pattern matching algorithm we need to answer

Where are the matched patterns?

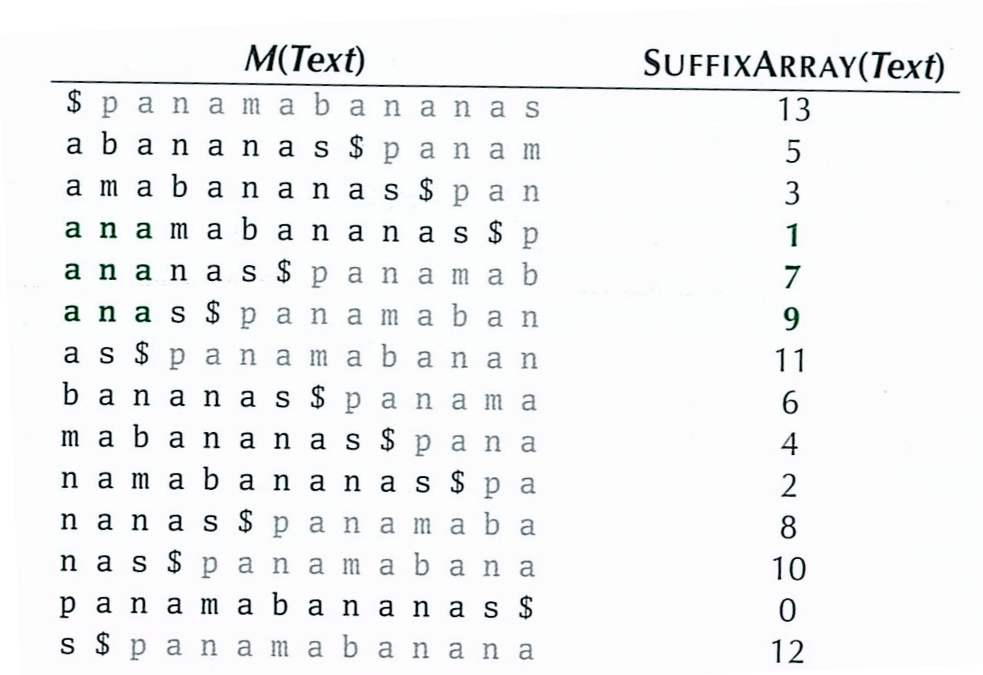
and

How do we deal the mis-match tolerance?

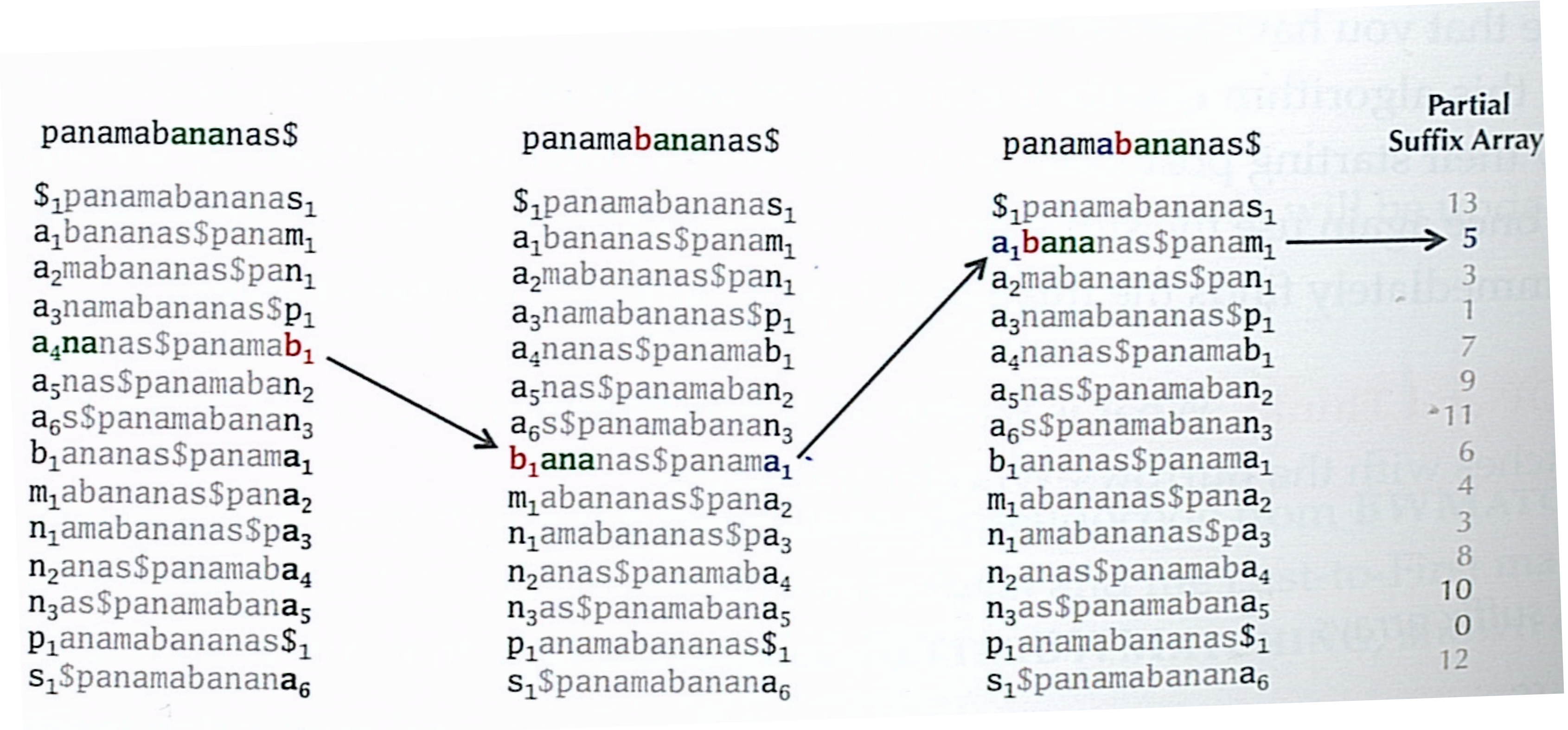
Where are the matched patterns?

The suffixes of **`panamabananas$`** that begin the rows of M(**`panamabananas$`**) are highlighted, and the suffixes beginning with **`ana`** are shown in green.

The suffix array records the starting position of each suffix in *Text*.



M(*Text*)	SUFFIXARRAY(*Text*)
$ p a n a m a b a n a n a s	13
a b a n a n a s $ p a n a m	5
a m a b a n a n a s $ p a n	3
a n a m a b a n a n a s $ p	1
a n a n a s $ p a n a m a b	7
a n a s $ p a n a m a b a n	9
a s $ p a n a m a b a n a n	11
b a n a n a s $ p a n a m a	6
m a b a n a n a s $ p a n a	4
n a m a b a n a n a s $ p a	2
n a n a s $ p a n a m a b a	8
n a s $ p a n a m a b a n a	10
p a n a m a b a n a n a s $	0
s $ p a n a m a b a n a n a	12



panamabananas$

$\$_1\text{panamabanana}s_1$
$a_1\text{bananas\$pana}m_1$
$a_2\text{mabananas\$pa}n_1$
$a_3\text{namabananas\$}p_1$
$a_4\text{nanas\$panama}b_1$
$a_5\text{nas\$panamaba}n_2$
$a_6\text{s\$panamabana}n_3$
$b_1\text{ananas\$panam}a_1$
$m_1\text{abananas\$pan}a_2$
$n_1\text{amabananas\$p}a_3$
$n_2\text{anas\$panamab}a_4$
$n_3\text{as\$panamaban}a_5$
$p_1\text{anamabananas}\$_1$
$s_1\text{\$panamabanan}a_6$

panamabananas$

$\$_1\text{panamabanana}s_1$
$a_1\text{bananas\$pana}m_1$
$a_2\text{mabananas\$pa}n_1$
$a_3\text{namabananas\$}p_1$
$a_4\text{nanas\$panama}b_1$
$a_5\text{nas\$panamaba}n_2$
$a_6\text{s\$panamabana}n_3$
$b_1\text{ananas\$panam}a_1$
$m_1\text{abananas\$pan}a_2$
$n_1\text{amabananas\$p}a_3$
$n_2\text{anas\$panamab}a_4$
$n_3\text{as\$panamaban}a_5$
$p_1\text{anamabananas}\$_1$
$s_1\text{\$panamabanan}a_6$

panamabananas$

	Partial Suffix Array
$\$_1\text{panamabanana}s_1$	13
$a_1\text{bananas\$pana}m_1$	5
$a_2\text{mabananas\$pa}n_1$	3
$a_3\text{namabananas\$}p_1$	1
$a_4\text{nanas\$panama}b_1$	7
$a_5\text{nas\$panamaba}n_2$	9
$a_6\text{s\$panamabana}n_3$	11
$b_1\text{ananas\$panam}a_1$	6
$m_1\text{abananas\$pan}a_2$	4
$n_1\text{amabananas\$p}a_3$	3
$n_2\text{anas\$panamab}a_4$	8
$n_3\text{as\$panamaban}a_5$	10
$p_1\text{anamabananas}\$_1$	0
$s_1\text{\$panamabanan}a_6$	12

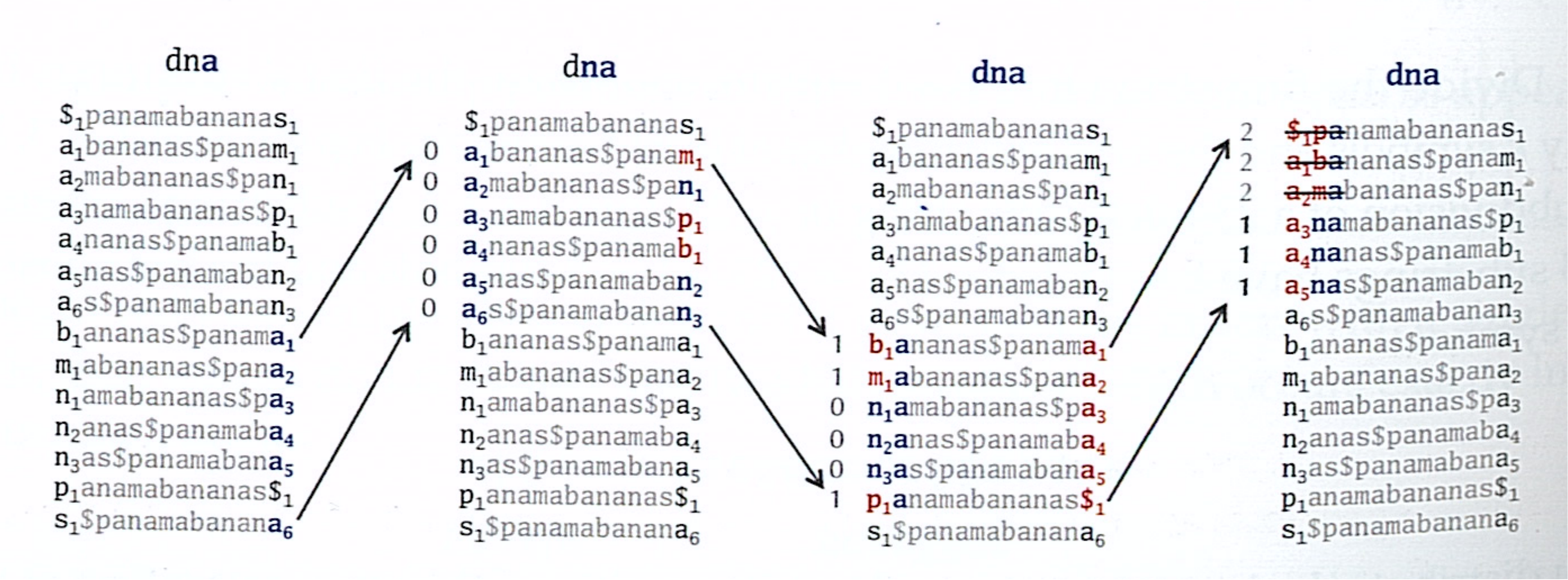
How do we deal the mis-match tolerance?

Pattern acttaggctcgggataatcc

Text actaagtctcgggataagcc

Theorem. If two strings of length n match with at most d mismatches, then they must share at least one k-mer of length k = [n/(d+1)]

To extend the BWT to approximate pattern matching, we will not stop when we encounter a mismatch. We proceed onward until we either find an approximate match or exceed the limit of *d* mismatches.



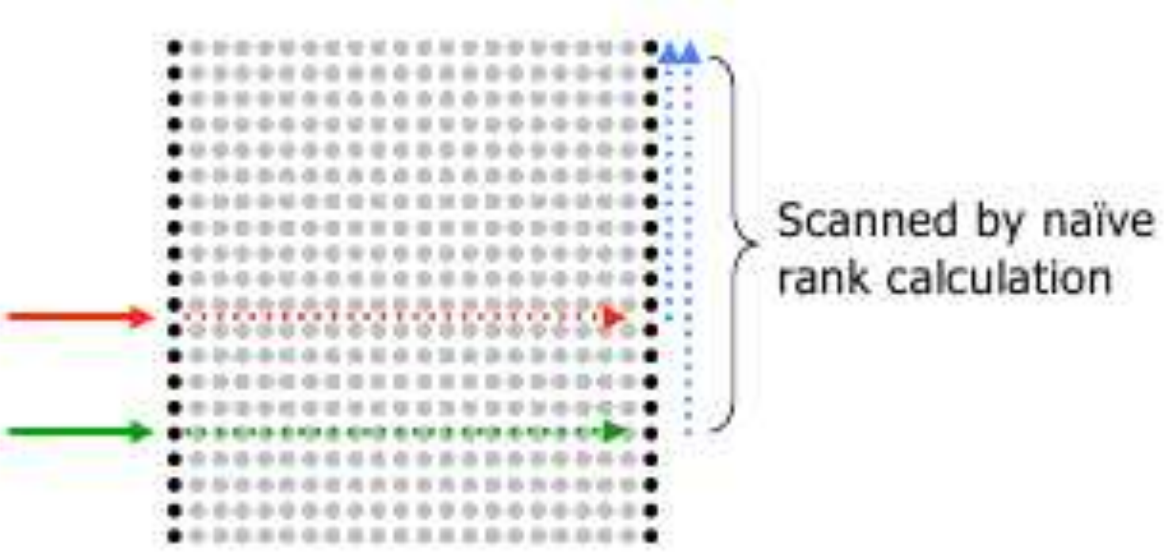
The number of mismatches encountered in a given row is shown in the column on the left.

Key properties of Burrows-Wheeler Transform

- **Very little memory usage. Same as input (or less)**
  - Don’t represent matrix, or strings, just pointers
  - Encode: Simply sort pointers. Decode: follow pointers
- **Original application: string compression (bZip2)**
  - Runs of letters compressed into (letter, runlength) pairs
- **Bioinformatics applications: substring searching**
  - Achieve similar run time as hash tables, suffix trees
  - But: very memory efficient ➔ practical speed gains
- **Mapping 100,000s of reads: only transform once**
  - Pre-process once; read counts in transformed space.
  - Reverse transform once, map counts to genome coords

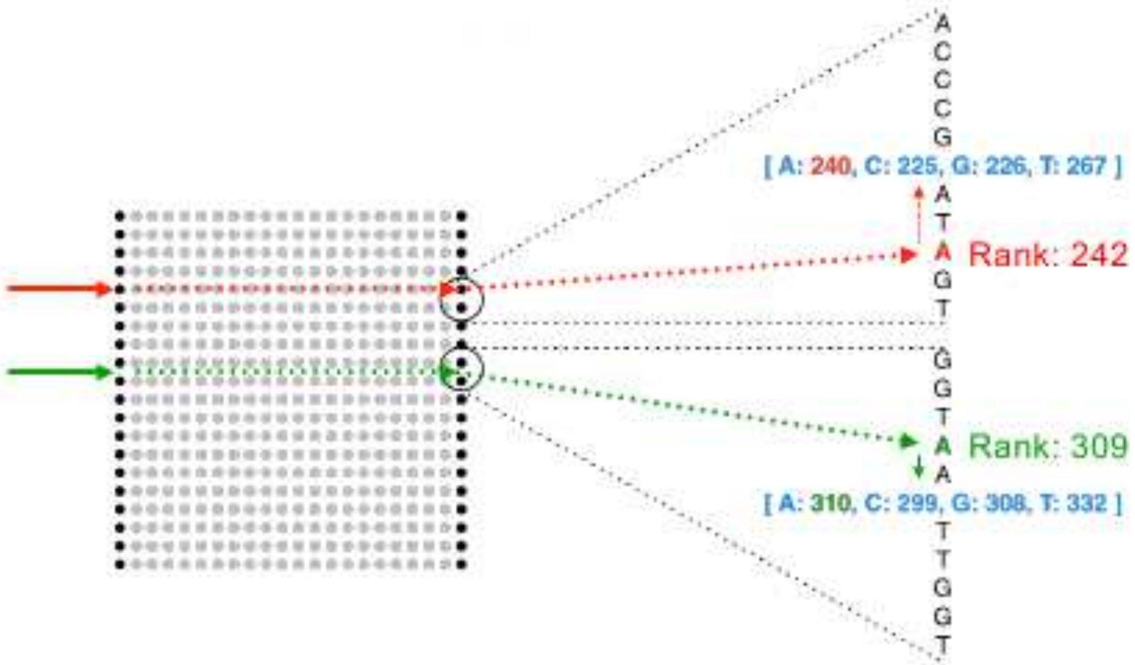
Checkpointing in FM Index

- $\mathbf{LF}(i, c)$ determines the rank of qc in row i
- Nave way: count occurrences of qc in all previous rows, complexity is linear in length of text too slow



Checkpointing in FM Index

- Solution: pre-calculate cumulative counts for A/C/G/T up to periodic checkpoints in BWT



A
C
C
C
G
[A: 240, C: 225, G: 226, T: 267]
A
T
A
Rank: 242
G
T
G
G
T
A
Rank: 309
A
[A: 310, C: 299, G: 308, T: 332]
T
T
G
G
T

Relationship Between BWT and Suffix Arrays

s = appellee$
    123456789

$appellee	$	9	s[9-1] = e
appellee$	appellee$	1	s[1-1] = $
e$appelle	e$	8	s[8-1] = e
ee$appell	ee$	7	s[7-1] = l
ellee$app	ellee$	4	s[4-1] = p
lee$appel	lee$	6	s[6-1] = l
llee$appe	llee$	5	s[5-1] = e
pellee$ap	pellee$	3	s[3-1] = p
ppellee$a	ppellee$	2	s[2-1] = a

These are still in sorted order because "$" comes before everything else

– subtract 1 →

BWT matrix

The suffixes are obtained by deleting everything after the $

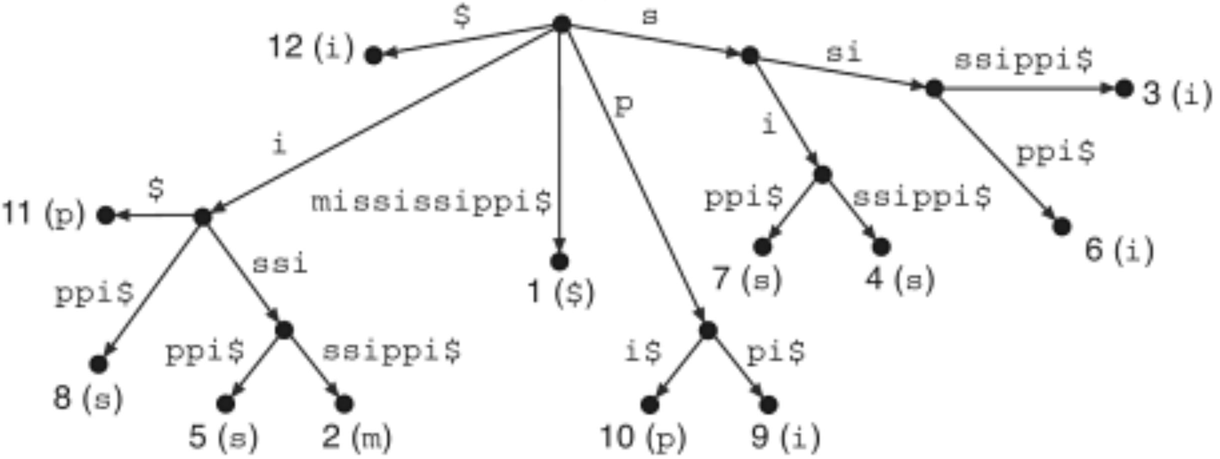
Suffix array (start position for the suffixes)

Suffix position - 1 = the position of the last character of the BWT matrix

($ is a special case)

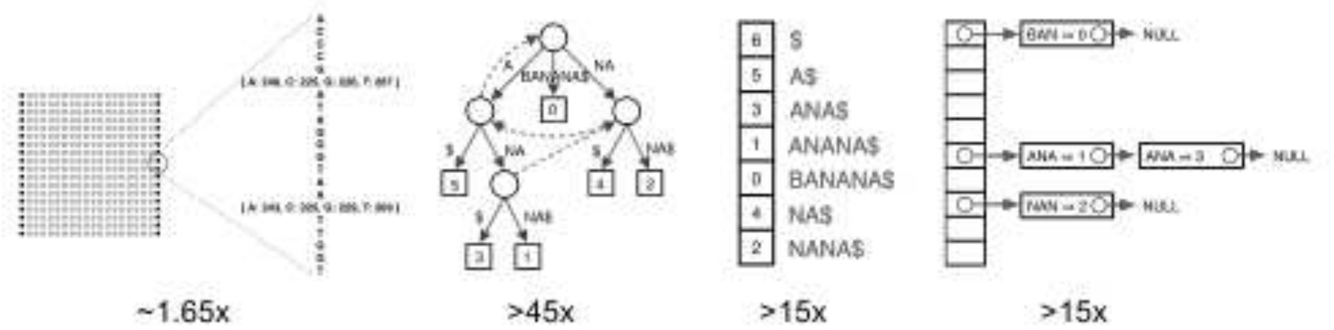
Relationship Between BWT and Suffix Trees

Suffixes	ID	Sorted Suffixes	Suffix Array	Sorted Rotations (A_s matrix)	BWT Output (L)
mississippi$	1	$	12	$mississippi	i
ississippi$	2	i$	11	i$mississipp	p
ssissippi$	3	ippi$	8	ippi$mississ	s
sissippi$	4	issippi$	5	issippi$miss	s
issippi$	5	ississippi$	2	ississippi$m	m
ssippi$	6	mississippi$	1	mississippi$	$
sippi$	7	pi$	10	pi$mississip	p
ippi$	8	ppi$	9	ppi$mississi	i
ppi$	9	sippi$	7	sippi$missis	s
pi$	10	sissippi$	4	sissippi$mis	s
i$	11	ssippi$	6	ssippi$missi	i
$	12	ssissippi$	3	ssissippi$mi	i

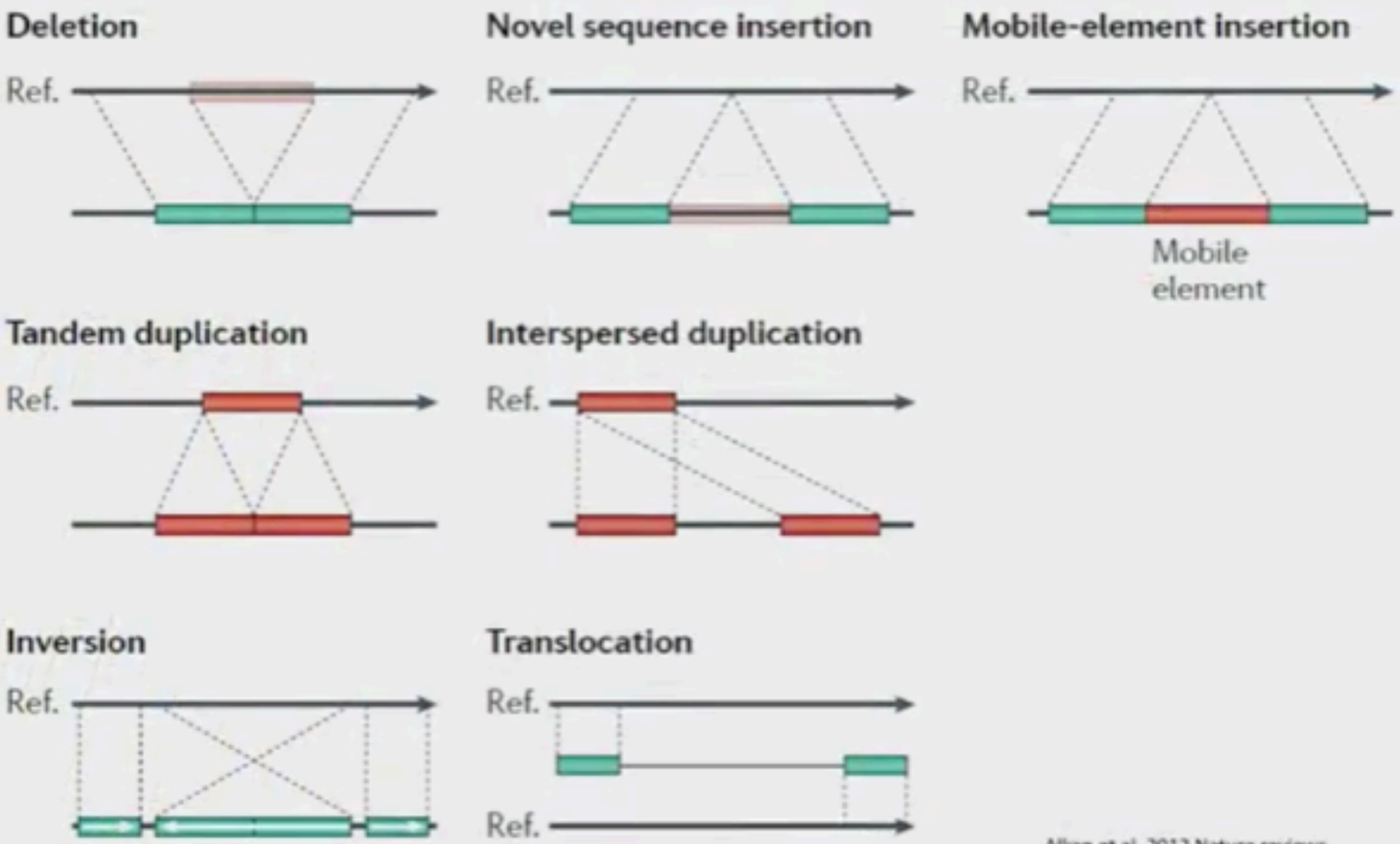


FM Index is Small

- Entire FM Index on DNA reference consists of:
  - BWT (same size as x)
  - Checkpoints (15% size of x)
  - SA sample (50% size of x)
- Total: 1.65x the size of x

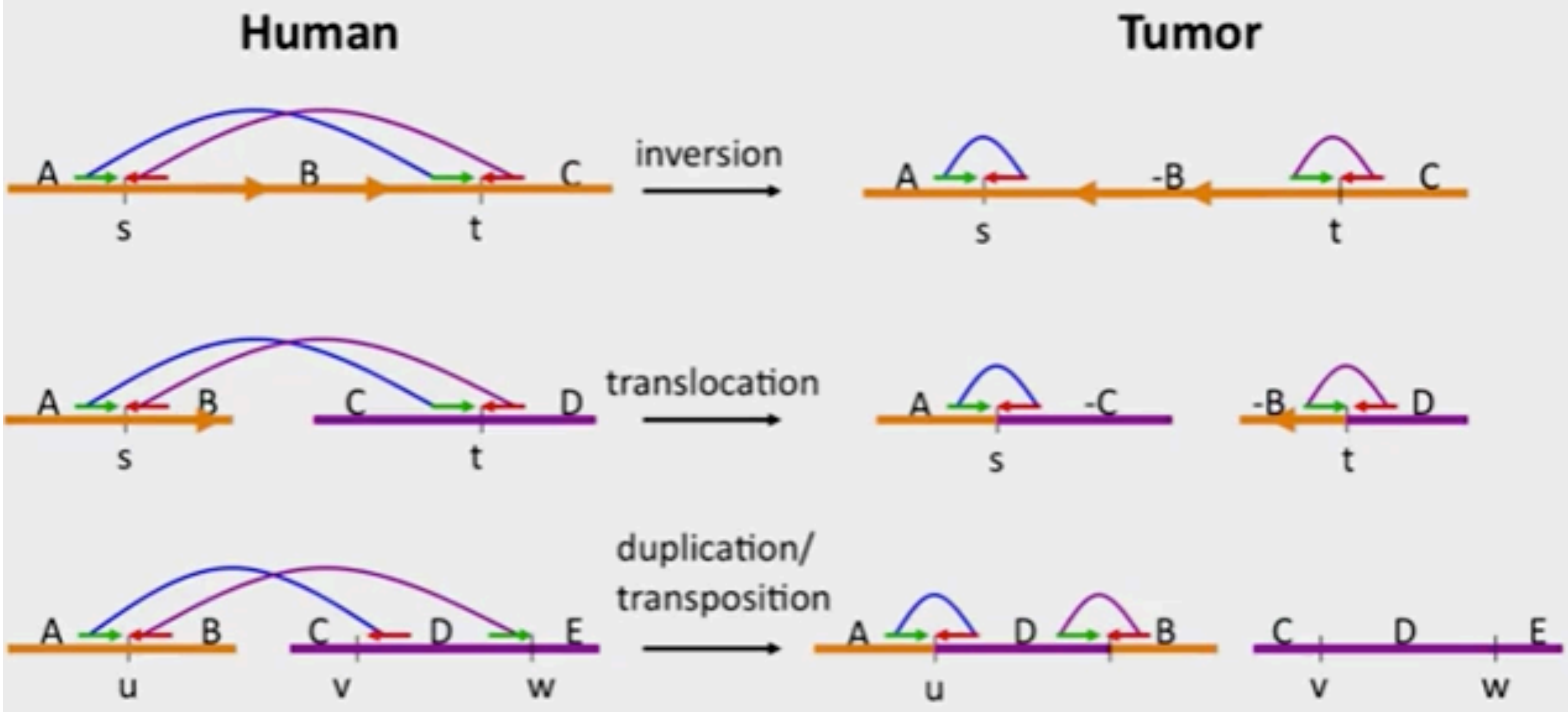


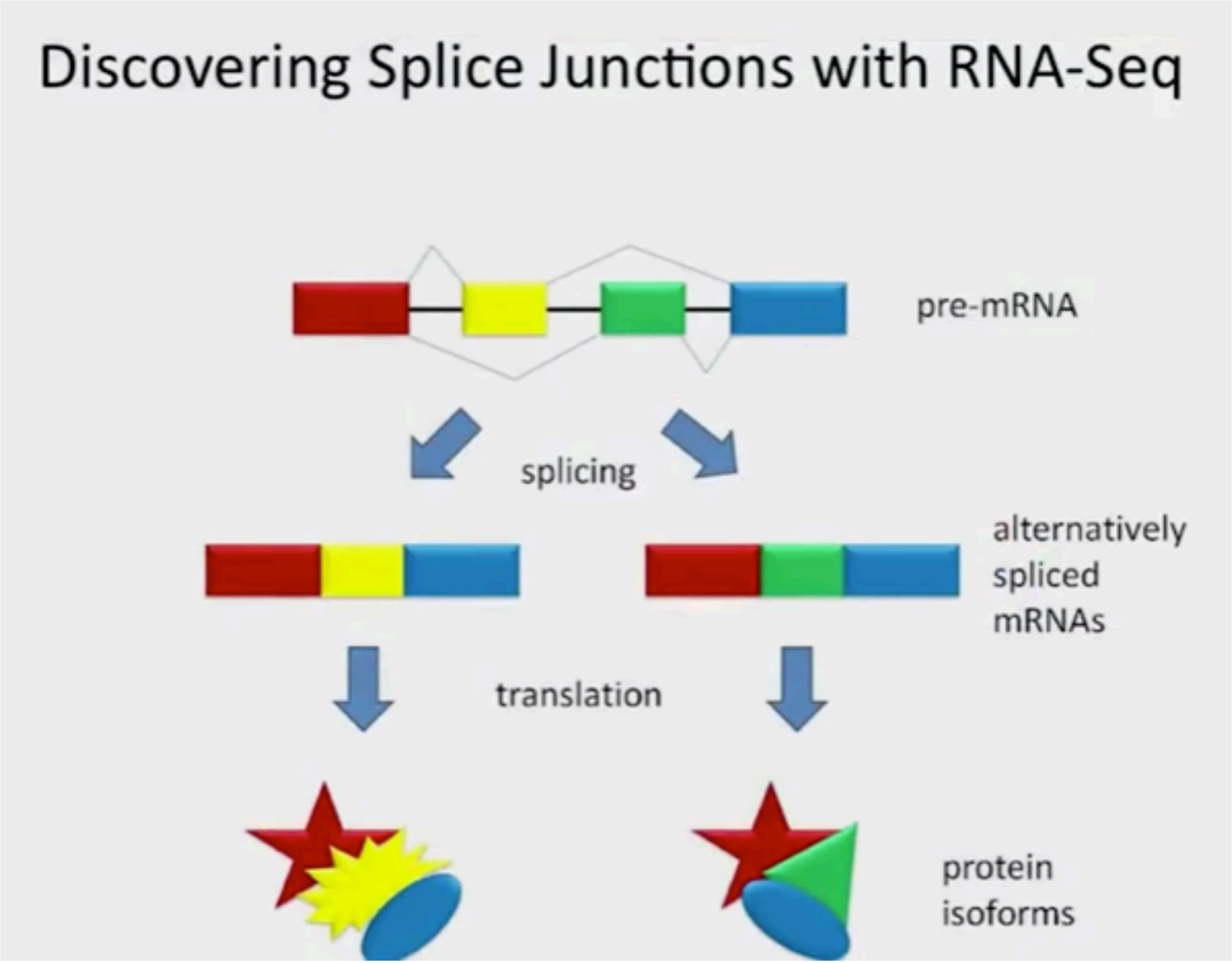
Structure Variations



Alkan et al, 2012 Nature reviews

Variant Calling by Read Mapping





Discovering Splice Junctions with RNA-Seq

pre-mRNA

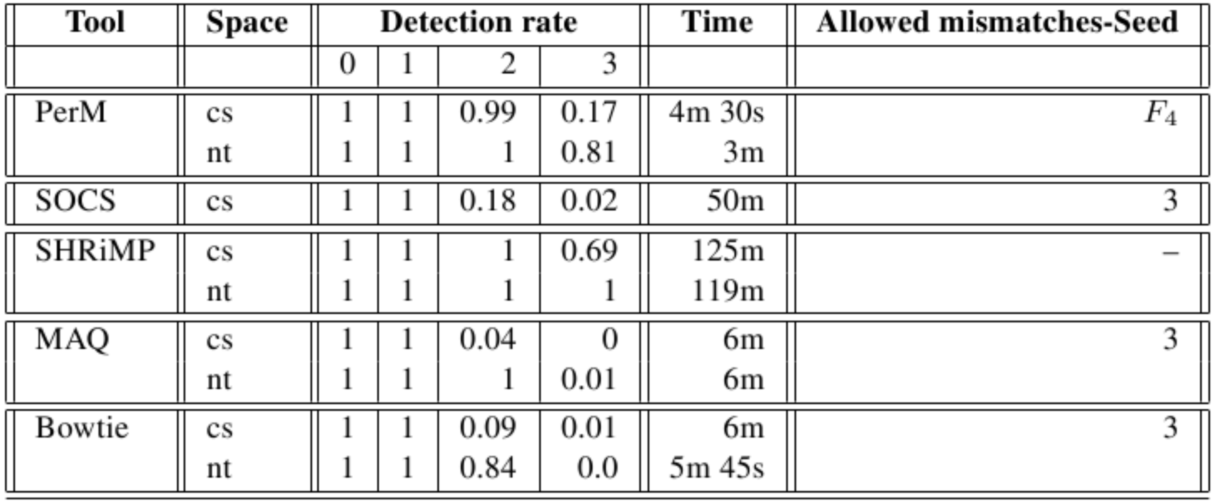
splicing

alternatively spliced mRNAs

translation

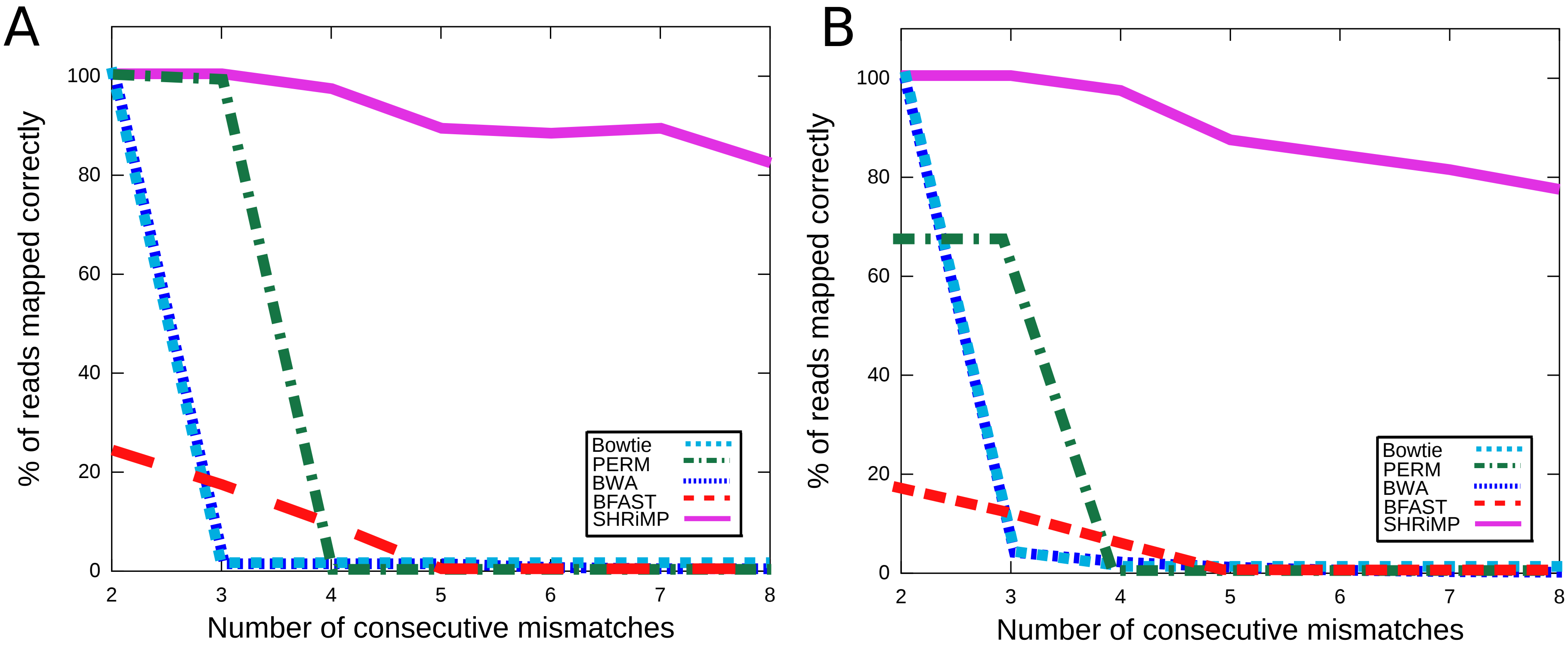
protein isoforms

Algorithms comparison



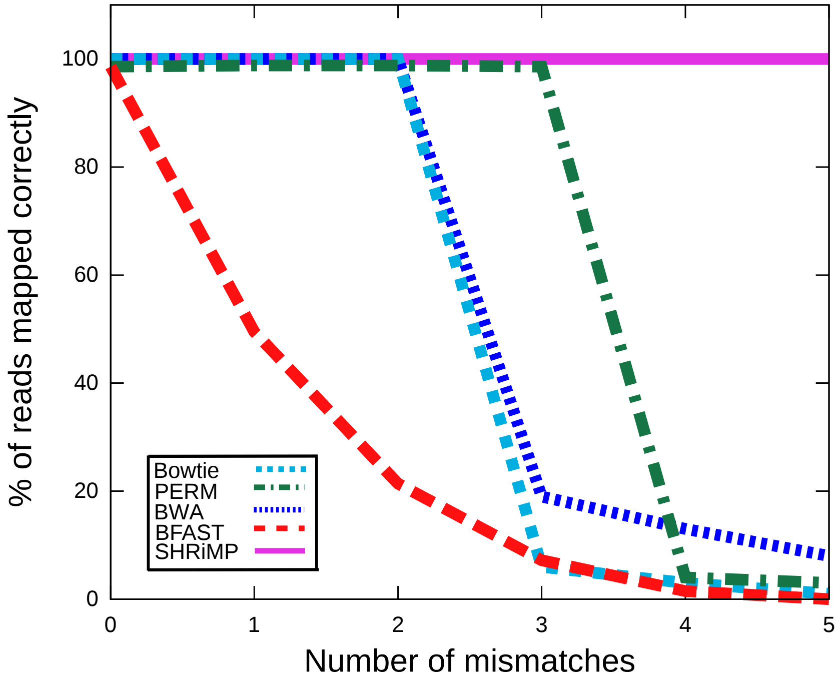
Tool	Space	Detection rate				Time	Allowed mismatches-Seed
		0	1	2	3		
PerM	cs	1	1	0.99	0.17	4m 30s	F_4
	nt	1	1	1	0.81	3m	
SOCS	cs	1	1	0.18	0.02	50m	3
SHRiMP	cs	1	1	1	0.69	125m	–
	nt	1	1	1	1	119m	
MAQ	cs	1	1	0.04	0	6m	3
	nt	1	1	1	0.01	6m	
Bowtie	cs	1	1	0.09	0.01	6m	3
	nt	1	1	0.84	0.0	5m 45s	

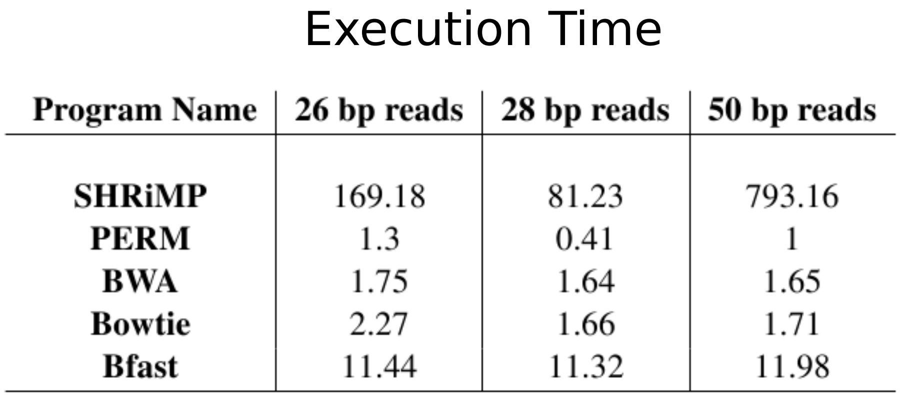
Algorithms comparison



The accuracy of the various tools considering a synthetic dataset with continuous mismatches.

Algorithms comparison





Execution Time

Program Name	26 bp reads	28 bp reads	50 bp reads
SHRiMP	169.18	81.23	793.16
PERM	1.3	0.41	1
BWA	1.75	1.64	1.65
Bowtie	2.27	1.66	1.71
Bfast	11.44	11.32	11.98

Accuracy of the various tools over synthetic dataset and the execution time of each tool.