Burrows-Wheeler Transform

CMSC 423

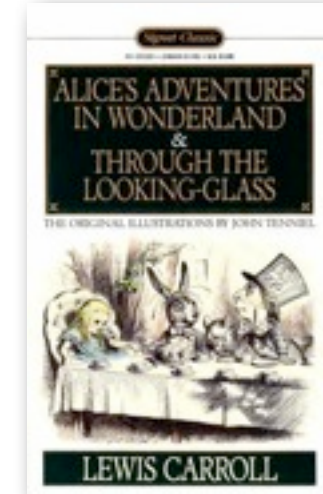
Genome of the Cow

a sequence of 2.86 billion letters

enough letters to fill a million pages of a typical book.

```
TATGGAGCCAGGTGCCTGGGGCAACAAGACTGTGGTCACTGAATTCATCCTTCTTGGTCTAACAGAGAACATAG
AACTGCAATCCATCCTTTTTGCCATCTTCCTCTTTGCCTATGTGATCACAGTCGGGGGCAACTTGAGTATCCTG
GCCGCCATCTTTGTGGAGCCCAAACTCCACACCCCCATGTACTACTTCCTGGGGAACCTTTCTCTGCTGGACAT
TGGGTGCATCACTGTCACCATTCCTCCCATGCTGGCCTGTCTCCTGACCCACCAATGCCGGGTTCCCTATGCAG
CCTGCATCTCACAGCTCTTCTTTTTCCACCTCCTGGCTGGAGTGGACTGTCACCTCCTGACAGCCATGGCCTAC
GACCGCTACCTGGCCATTTGCCAGCCCCTCACCTATAGCATCCGCATGAGCCGTGACGTCCAGGGAGCCCTGGT
GGCCGTCTGCTGCTCCATCTCCTTCATCAATGCTCTGACCCACACAGTGGCTGTGTCTGTGCTGGACTTCTGCG
GCCCTAACGTGGTCAACCACTTCTACTGTGACCTCCCGCCCCTTTTCCAGCTCTCCTGCTCCAGCATCCACCTC
AACGGGCAGCTACTTTTCGTGGGGGCCACCTTCATGGGGGTGGTCCCCATGGTCTTCATCTCGGTATCCTATGC
CCACGTGGCAGCCGCAGTCCTGCGGATCCGCTCGGCAGAGGGCAGGAAGAAAGCCTTCTCCACGTGTGGCTCCC
ACCTCACCGTGGTCTGCATCTTTTATGGAACCGGCTTCTTCAGCTACATGCGCCTGGGCTCCGTGTCCGCCTCA
GACAAGGACAAGGGCATTGGCATCCTCAACACTGTCATCAGCCCCATGCTGAACCCACTCATCTACAGCCTCCG
GAACCCTGATGTGCAGGGCGCCCTGAAGAGGTTGCTGACAGGGAAGCGGCCCCCGGAGTG ...
```

We can only read ~ 1000 characters at a time from a random place:




```
good-natured, she thought: still
                    when it saw Alice. It looked
        ought to be treated
                         good-natured, she thought, still
                     Cat only
                    a greet many
                                        It looked good-
 The Cat only grinned when it saw Alice.
     be treated with respect.
                        still it had very long claws
           claws and a great many teeth, so she

                                 so she felt that it ought
```

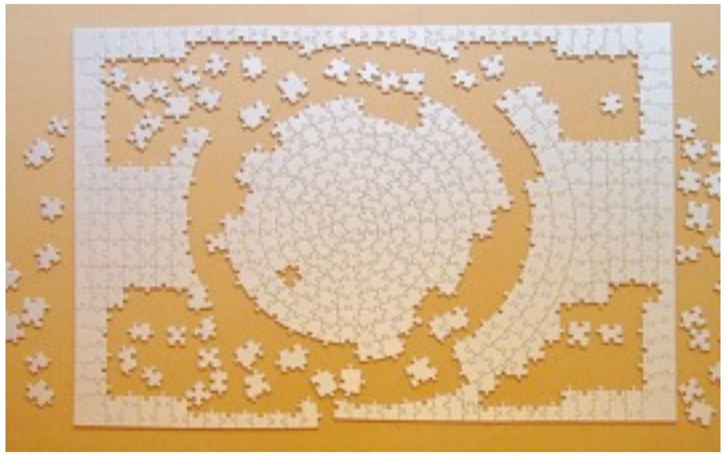
```
The Cat only grinned when it saw Alice.
    Cat only         when it saw Alice. It looked
                                        It looked good-

good-natured, she thought: still
good-natured, she thought, still
                           still it had very long claws

claws and a great many teeth, so she
          a greet many        so she felt that it ought

ought to be treated
         be treated with respect.
```

It's a jigsaw puzzle ...



...except with 35 million pieces

Motivation - Short Read Mapping

Sequencing technologies produce millions of "reads" = a random, short substring of the genome

If we already know the genome of one cow, we can get reads from a 2nd cow and map them onto the known cow genome.

Need to do millions of string searches in a long string.

Burrows-Wheeler Transform

Text transform that is useful for compression & search.

banana

banana$		$banana
anana$b		a$banan
nana$ba	sort →	ana$ban
ana$ban		anana$b
na$bana		banana$
a$banan		nana$ba
$banana		na$bana

BWT(banana) =
annb$aa

Tends to put runs of the same character together.

Makes compression work well.

"bzip" is based on this.

Another Example

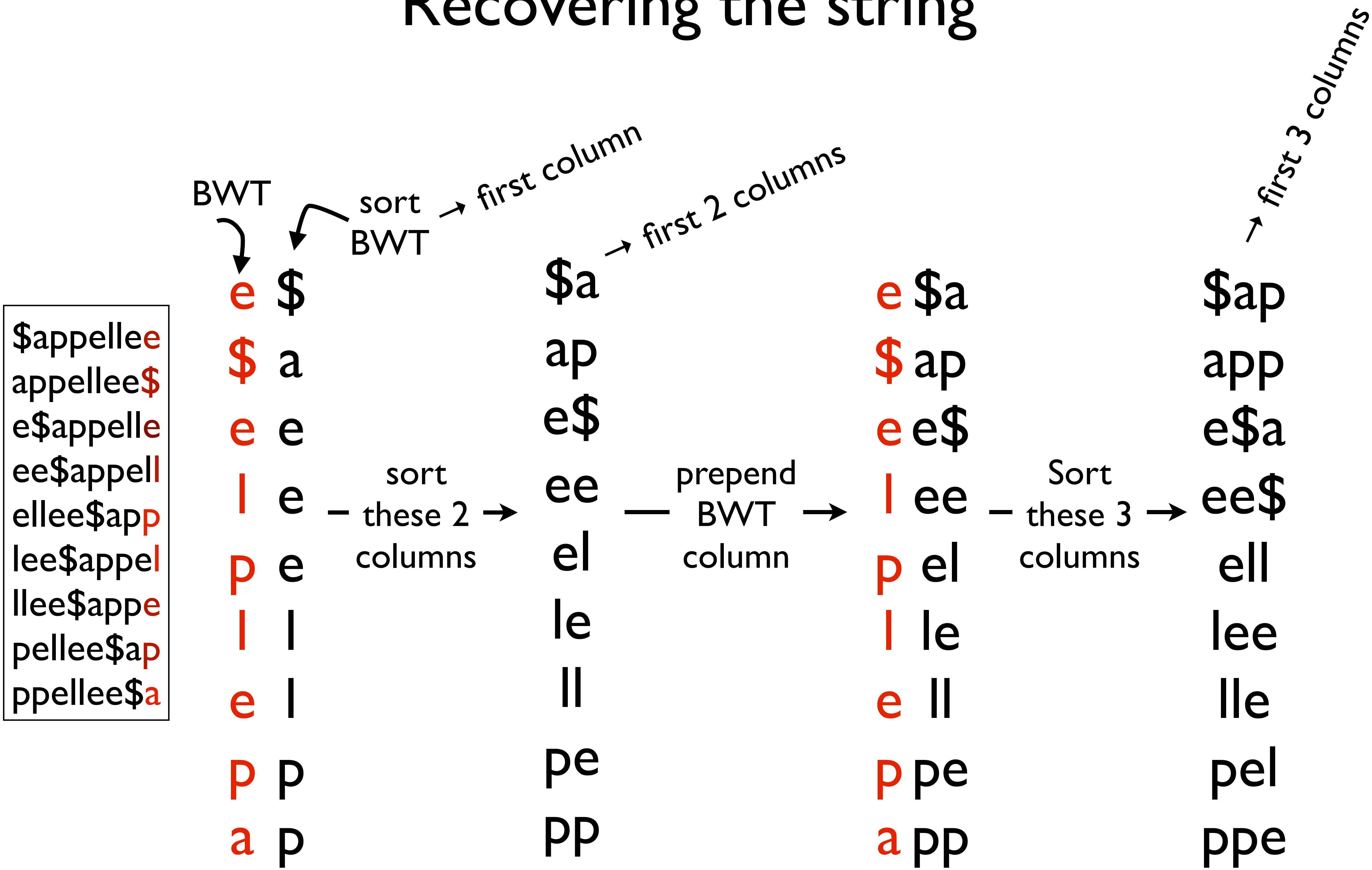
appellee$

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

BWT(appellee$) =
e$elplepa

Doesn't always improve the compressibility...

Recovering the string



Inverse BWT

```
def inverseBWT(s):
   B = [s1,s2,s3,...,sn]
   for i = 1..n:
      sort B
      prepend si to B[i]
   return row of B that ends with $
```

Another BWT Example

Rotations		Sorted rotations		
dogwood$		$dogwood		
ogwood$d		d$dogwoo		
gwood$do		dogwood$	last column →	
wood$dog	sort →	gwood$do		
ood$dogw		od$dogwo		BWT(dogwood$) =
od$dogwo		ogwood$d		do$oodwg
d$dogwoo		ood$dogw		
$dogwood		wood$dog		

do$oodwg

Another BWT Example

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

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

Searching with BWT: LF Mapping

LF Mapping

Σ

BWT(unabashable)	$	a	b	e	h	l	n	s	u
$unabashable	0	0	0	0	0	0	0	0	0
ababashable$un	0	0	0	1	0	0	0	0	0
able$unabash	0	0	0	1	0	0	1	0	0
ashable$unab	0	0	0	1	1	0	1	0	0
bashable$una	0	0	1	1	1	0	1	0	0
ble$unabasha	0	1	1	1	1	0	1	0	0
e$unabashabl	0	2	1	1	1	0	1	0	0
hable$unabas	0	2	1	1	1	1	1	0	0
le$unabashab	0	2	1	1	1	1	1	1	0
nabashable$u	0	2	2	1	1	1	1	1	0
shable$unaba	0	2	2	1	1	1	1	1	1
unabashable$	0	3	2	1	1	1	1	1	1
	1	3	2	1	1	1	1	1	1

# of times letter appears before this position in the last column.

LF Property: The i-th occurrence of a letter X in the last column corresponds to the i-th occurrence of X in the first column.

BWT Search

BWTSearch(aba)   Start from the **end** of the pattern

Step 1: Find the range of "a"s in the first column

Step 2: Look at the same range in the last column.

Step 3: "b" is the next pattern character. Set B = the LF mapping entry for b in the first row of the range.
Set E = the LF mapping entry for b in the last + 1 row of the range.

Step 4: Find the range for "b" in the first row, and use B and E to find the right subrange within the "b" range.

BWT(unabashable)

$unabashable
abashable$un
able$unabash
ashable$unab
bashable$una
ble$unabasha
e$unabashabl
hable$unabas
le$unabashab
nabashable$u
shable$unaba
unabashable$

LF Mapping

Σ

$	a	b	e	h	l	n	s	u
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	0	1	0	0	1	0	0
0	0	0	1	1	0	1	0	0
0	0	1	1	1	0	1	0	0
0	1	1	1	1	0	1	0	0
0	2	1	1	1	0	1	0	0
0	2	1	1	1	1	1	0	0
0	2	1	1	1	1	1	1	0
0	2	2	1	1	1	1	1	0
0	2	2	1	1	1	1	1	1
0	3	2	1	1	1	1	1	1
1	3	2	1	1	1	1	1	1

BWT Searching Example 2

pattern = “bana”

a	$	a	b	n
$bananna	0	0	0	0
a$banann	0	1	0	0
ananna$b	0	1	0	1
anna$ban	0	1	1	1
bananna$	0	1	1	2
na$banan	1	1	1	2
nanna$ba	1	1	1	3
nna$bana	1	2	1	3
	1	3	1	3

n	$	a	b	n
$bananna	0	0	0	0
a$banann	0	1	0	0
ananna$b	0	1	0	1
anna$ban	0	1	1	1
bananna$	0	1	1	2
na$banan	1	1	1	2
nanna$ba	1	1	1	3
nna$bana	1	2	1	3
	1	3	1	3

(B,E) = 0, 2

n	$	a	b	n
$bananna	0	0	0	0
a$banann	0	1	0	0
ananna$b	0	1	0	1
anna$ban	0	1	1	1
bananna$	0	1	1	2
na$banan	1	1	1	2
nanna$ba	1	1	1	3
nna$bana	1	2	1	3
	1	3	1	3

a	$	a	b	n
$bananna	0	0	0	0
a$banann	0	1	0	0
ananna$b	0	1	0	1
anna$ban	0	1	1	1
bananna$	0	1	1	2
na$banan	1	1	1	2
nanna$ba	1	1	1	3
nna$bana	1	2	1	3
	1	3	1	3

(B,E) = 1, 2

a	$	a	b	n
$bananna	0	0	0	0
a$banann	0	1	0	0
ananna$b	0	1	0	1
anna$ban	0	1	1	1
bananna$	0	1	1	2
na$banan	1	1	1	2
nanna$ba	1	1	1	3
nna$bana	1	2	1	3
	1	3	1	3

(B,E) = 0, 1

b	$	a	b	n
$bananna	0	0	0	0
a$banann	0	1	0	0
ananna$b	0	1	0	1
anna$ban	0	1	1	1
bananna$	0	1	1	2
na$banan	1	1	1	2
nanna$ba	1	1	1	3
nna$bana	1	2	1	3
	1	3	1	3

BWT Searching Notes

- Don't have to store the LF mapping. A more complex algorithm lets you compute it in O(1) time on the fly with only a little bit of storage.

- To find the range in the first column corresponding to a character:
  - Pre-compute array C[c] = # of occurrences in the string of characters lexicographically < c.
  - Then start of the "a" range, for example, is: C["a"] + 1.

- Running time: O(|pattern|)
  - Finding the range in the first column takes O(1) time using the C array.
  - Updating the range takes O(1) time using the LF mapping.

Relationship Between BWT and Suffix Arrays

```
s = appellee$
    123456789
```

BWT matrix	Suffixes	Suffix array	
$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

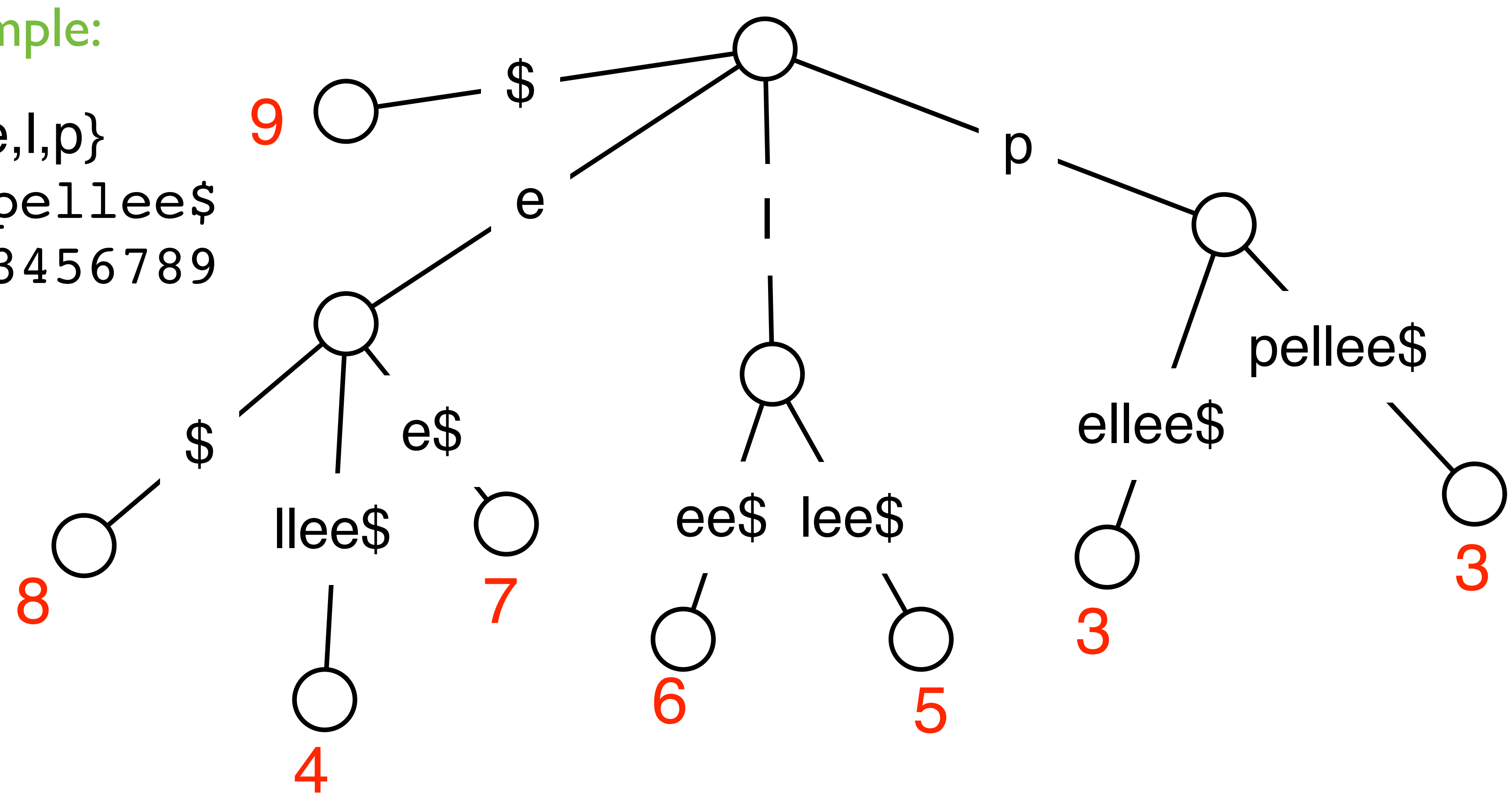
- Remember: Suffix Array = suffix numbers obtained by traversing the leaf nodes of the (ordered) Suffix Tree from left to right.
- Suffix Tree ⇒ Suffix Array ⇒ BWT.

Ordered suffix tree for previous example:

$\Sigma = \{\$, e, l, p\}$

s = `appellee$`

`123456789`



Computing BWT in O(n) time

- Easy $O(n^2 \log n)$-time algorithm to compute the BWT (create and sort the BWT matrix explicitly).
- Several direct O(n)-time algorithms for BWT. These are space efficient.
- Also can use suffix arrays or trees:

  Compute the suffix array, use correspondence between suffix array and BWT to output the BWT.

  O(n)-time and O(n)-space, but the constants are large.

Recap

BWT useful for searching and compression.

BWT is *invertible*: given the BWT of a string, the string can be reconstructed!

BWT is computable in O(n) time.

Close relationships between Suffix Trees, Suffix Arrays, and BWT:

- Suffix array = order of the suffix numbers of the suffix tree, traversed left to right
- BWT = letters at positions given by the suffix array entries - 1