Exact String Matching and searching for SNPs (2)

CMSC423

The problem

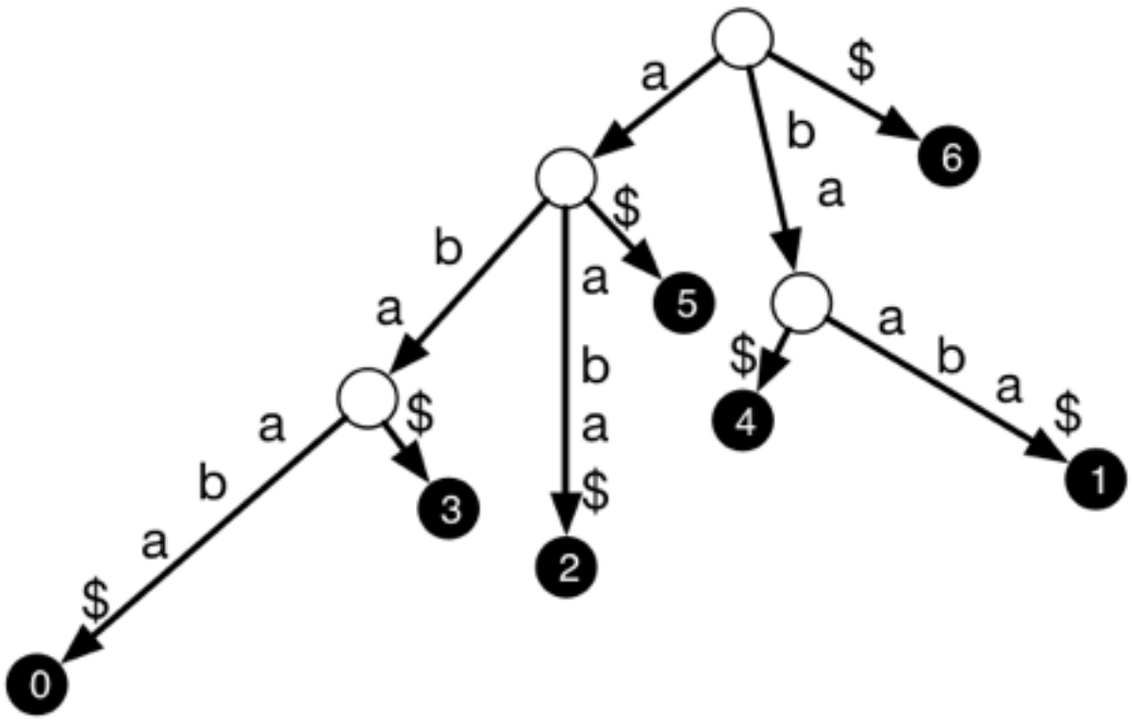
- Given:
  - 100's of millions of short reads: 100-200bp reads
  - A long reference genome (~3Bbp for human)
- Do:
  - Find high scoring scoring (fitting) alignments for each read
- What we know:
  - Dynamic programming solution for fitting alignment:
    - 1e8 * 1e9 * 1e2 operations, 1e9 * 1e2 memory

Strategies

- What if we only allow a small number of substitutions?
  - Let's first try to find *exact* matches and work from those (the d+1 trick in the midterm)
- We are aligning to the same reference 100's of millions of time
  - Is there preprocessing we can do to amortize time?
- Genomes are repetitive
  - Can we search for matches in the genome in a smart way?
  - Can we compress the genome, and search over the compressed representation?

Suffix Tree

T: abaaba$
0123456

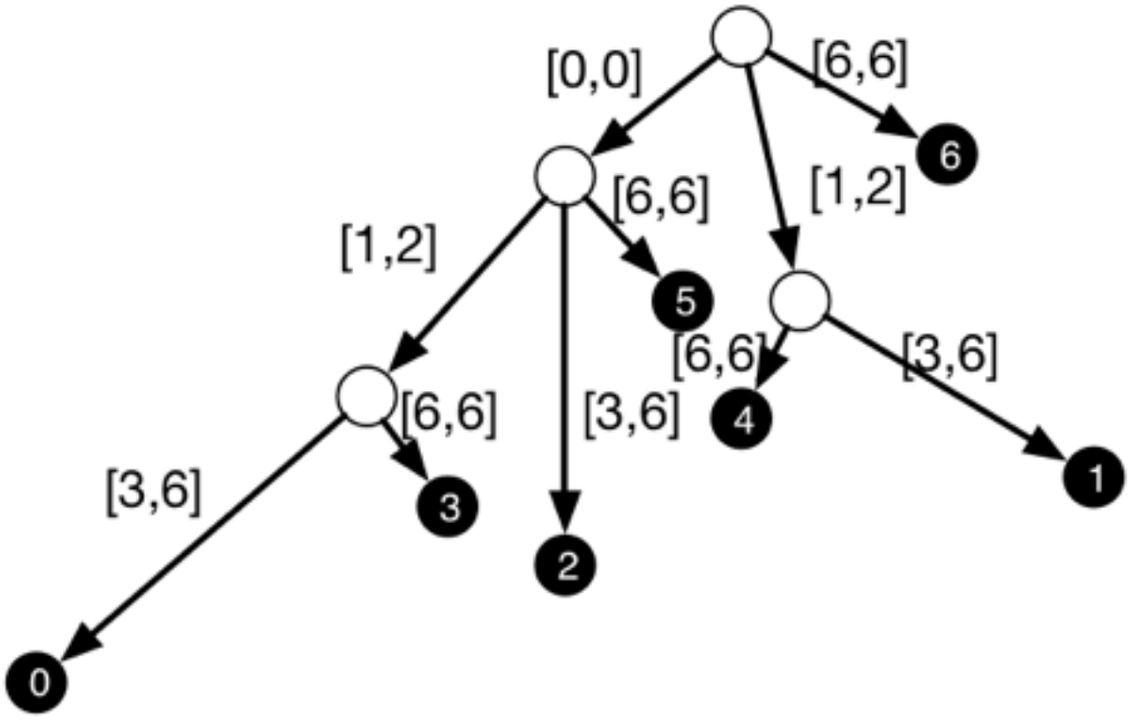


- Collapse non-branching nodes
  - #nodes O(|T|)

- Memory requirement is *not* O(|T|)
  - In the worst case, space required for edge labels is O(|T|)

Suffix Tree

T: abaaba$
0123456



- Collapse non-branching nodes
  - #nodes O(|T|)
- Label edges with substring *[start,end]*
  - O(1) per edge
- Memory now O(|T|)
- Construction algorithm O(|T|) (see Gusfield)

Recap

Structure	Processing Time	Memory	Search
Suffix Trie	O(|T|)	O(|T|^2)	O(|P|)
Suffix Tree	O(|T|)	O(|T|)*	O(|P|)
Suffix Array	O(|T|)	O(|T|) (but **much** smaller than Suffix Tree)	O(|P|log$_2$|T|)

*In best implementations about 20 bytes per character (as opposed to 4 bytes for suffix array)

Suffix Arrays

- Even though Suffix Trees are O(n) space, the constant hidden by the big-Oh notation is somewhat "big": ≈ 20 bytes / character in good implementations.

- If you have a 10Gb genome, 20 bytes / character = 200Gb to store your suffix tree. "Linear" but large.

- Suffix arrays are a more efficient way to store the suffixes that can do most of what suffix trees can do, but just a bit slower.

- Slight space vs. time tradeoff.

Example Suffix Array

s = attcatg$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

1	attcatg$
2	ttcatg$
3	tcatg$
4	catg$
5	atg$
6	tg$
7	g$
8	$

index of suffix suffix of s

sort the suffixes alphabetically

the indices just "come along for the ride"

8	$
5	atg$
1	attcatg$
4	catg$
7	g$
3	tcatg$
6	tg$
2	ttcatg$

Example Suffix Array

s = attcatg$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

1	attcatg$
2	ttcatg$
3	tcatg$
4	catg$
5	atg$
6	tg$
7	g$
8	$

index of suffix suffix of s

sort the suffixes alphabetically

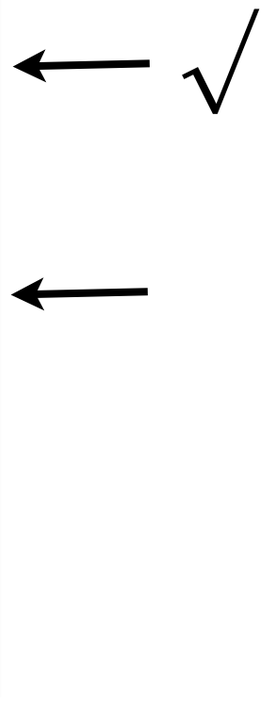
the indices just "come along for the ride"

8
5
1
4
7
3
6
2

Search via Suffix Arrays

s = cattcat$

8	$
6	at$
2	attcat$
5	cat$
1	cattcat$
7	t$
4	tcat$
3	ttcat$



- Does string "at" occur in s?
- Binary search to find "at".

- What about "tt"?

Counting via Suffix Arrays

s = cattcat$

8	$
6	at$
2	attcat$
5	cat$
1	cattcat$
7	t$
4	tcat$
3	ttcat$

- How many times does “at” occur in the string?
- All the suffixes that start with “at” will be next to each other in the array.
- Find one suffix that starts with “at” (using binary search).
- Then count the neighboring sequences that start with at.

Constructing Suffix Arrays

- Easy $O(n^2 \log n)$ algorithm:

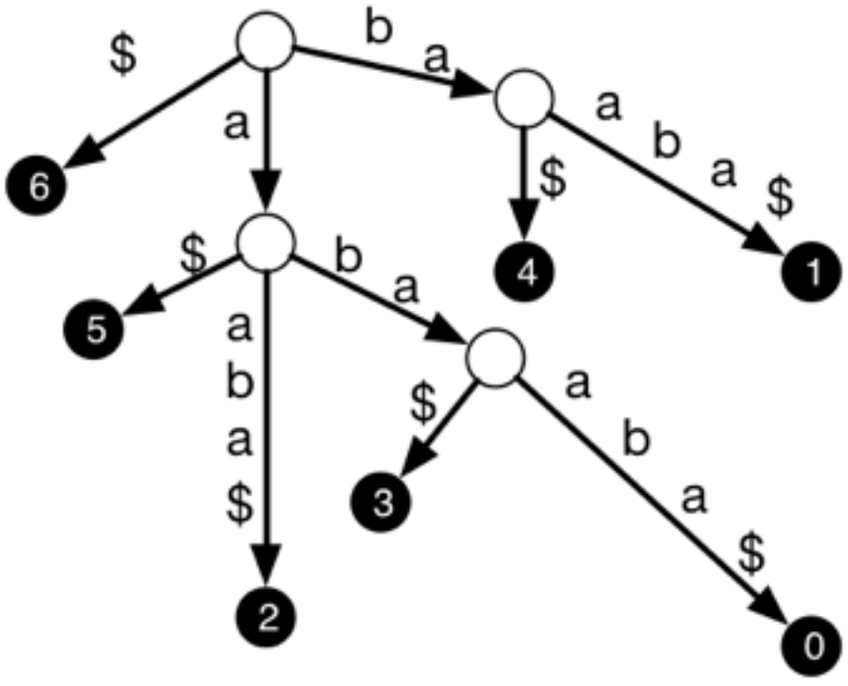
  sort the n suffixes, which takes O(n log n) comparisons, where each comparison takes O(n).

- There are several direct O(n) algorithms for constructing suffix arrays that use very little space.

- An simple O(n) algorithm: build the suffix tree, and exploit the relationship between suffix trees and suffix arrays (next slide)

Relationship between Suffix Arrays and Suffix Trees

T: abaaba$
0123456



6	$
5	a$
2	aaba$
3	aba$
0	abaaba$
4	ba$
1	baaa$

Build suffix trees with edge labels sorted lexicographically
Order of leaves: 6,5,2,3,0,4,1

Recap

Structure	Processing Time	Memory	Search
Suffix Trie	O(|T|)	O(|T|^2)	O(|P|)
Suffix Tree	O(|T|)	O(|T|)*	O(|P|)
Suffix Array	O(|T|)	O(|T|) (but **much** smaller than Suffix Tree)	O(|P|log$_2$|T|)

*In best implementations about 20 bytes per character (as opposed to 4 bytes for suffix array)

Burrows-Wheeler Transform

Text transform that is useful for compression & search.

banana

banana$	$banana	BWT(banana) =
anana$b	a$banan	annb$aa
nana$ba	ana$ban	
ana$ban	anana$b	
na$bana	banana$	
a$banan	nana$ba	
$banana	na$bana	

sort →

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

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

BWT → sort → BWT

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

sort these 2 columns → first column

$a
ap
e$
ee
el
le
ll
pe
pp

prepend BWT column → first 2 columns

e	$a
$	ap
e	e$
l	ee
p	el
l	le
e	ll
p	pe
a	pp

Sort these 3 columns → first 3 columns

$ap
app
e$a
ee$
ell
lee
lle
pel
ppe

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

dogwood$
ogwood$d
gwood$do
wood$dog
ood$dogw
od$dogwo
d$dogwoo
$dogwood

sort →

$dogwood
d$dogwoo
dogwood$
gwood$do
od$dogwo
ogwood$d
ood$dogw
wood$dog

last column →

BWT(dogwood$) =
do$oodwg

do$oodwg Another BWT Example

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

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

Searching with BWT: LF Mapping

LF Mapping

BWT(unabashable)	Σ								
	$	a	b	e	h	l	n	s	u
$unabashable	0	0	0	0	0	0	0	0	0
abashable$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)

	LF Mapping Σ								
	$	a	b	e	h	l	n	s	u
$unabashable	0	0	0	0	0	0	0	0	0
abashable$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

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 (later slides) lets you compute it in O(1) time in ***compressed*** data on the fly with some extra 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	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
$appellee	$	These are still in sorted order because "$" comes before everything else	9		s[9-1] = e
appellee$	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	– subtract 1 →	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

($ 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.

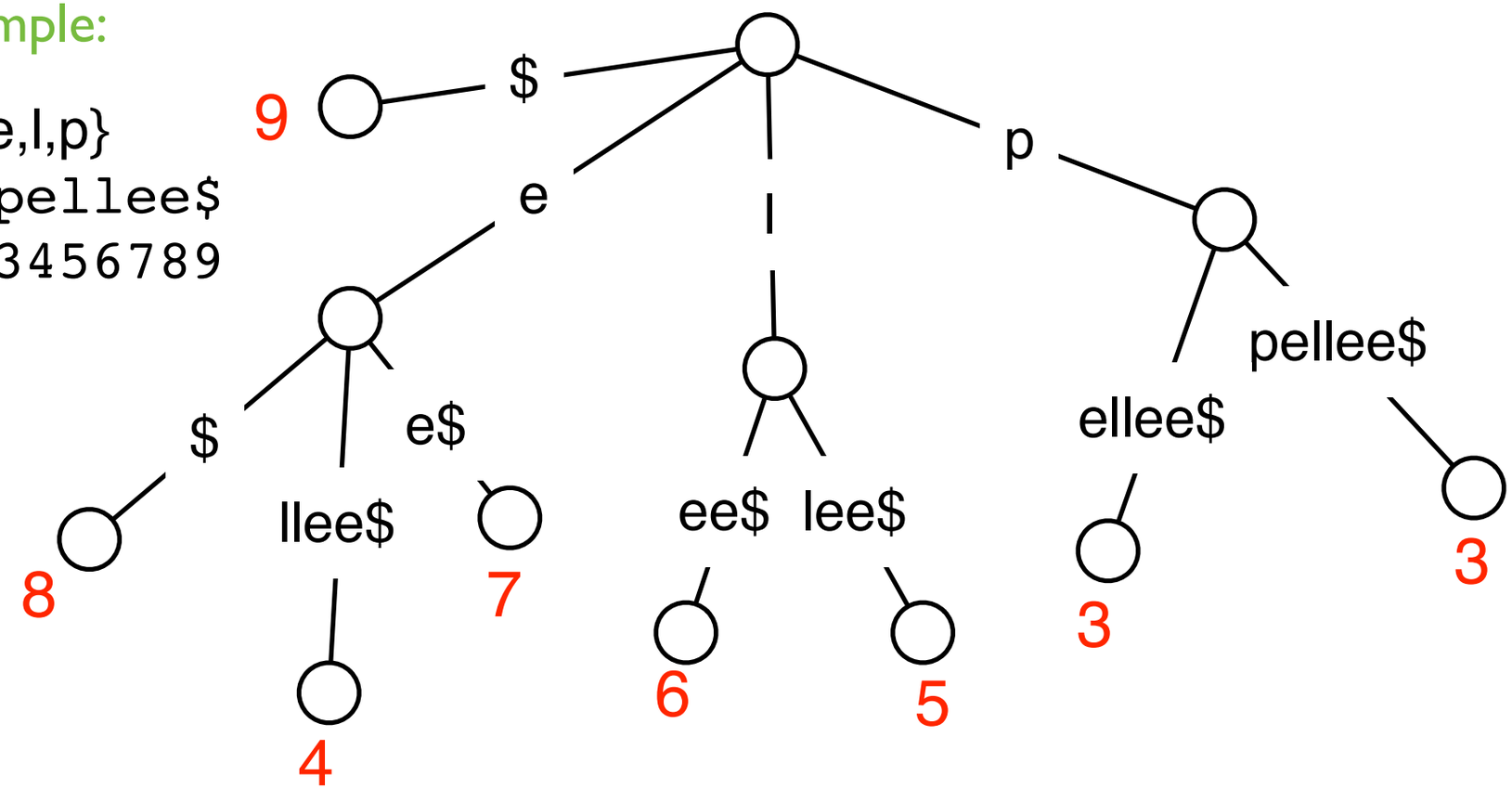
- Suffix Tree ⇒ Suffix Array ⇒ BWT.

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

Structure	Processing Time	Memory	Search
Suffix Trie	O(|T|)	O(|T|^2)	O(|P|)
Suffix Tree	O(|T|)	O(|T|)*	O(|P|)
Suffix Array	O(|T|)	O(|T|) (but **much** smaller than Suffix Tree)	O(|P|log$_2$|T|)
BWT	O(|T|)	O(|T|)**	O(|P|)

*In best implementations about 20 bytes per character (as opposed to 4 bytes for suffix array)
**Compressed! For human genome ~2GB