Exact String Matching and searching for SNPs (2)

CMSC423

The problem

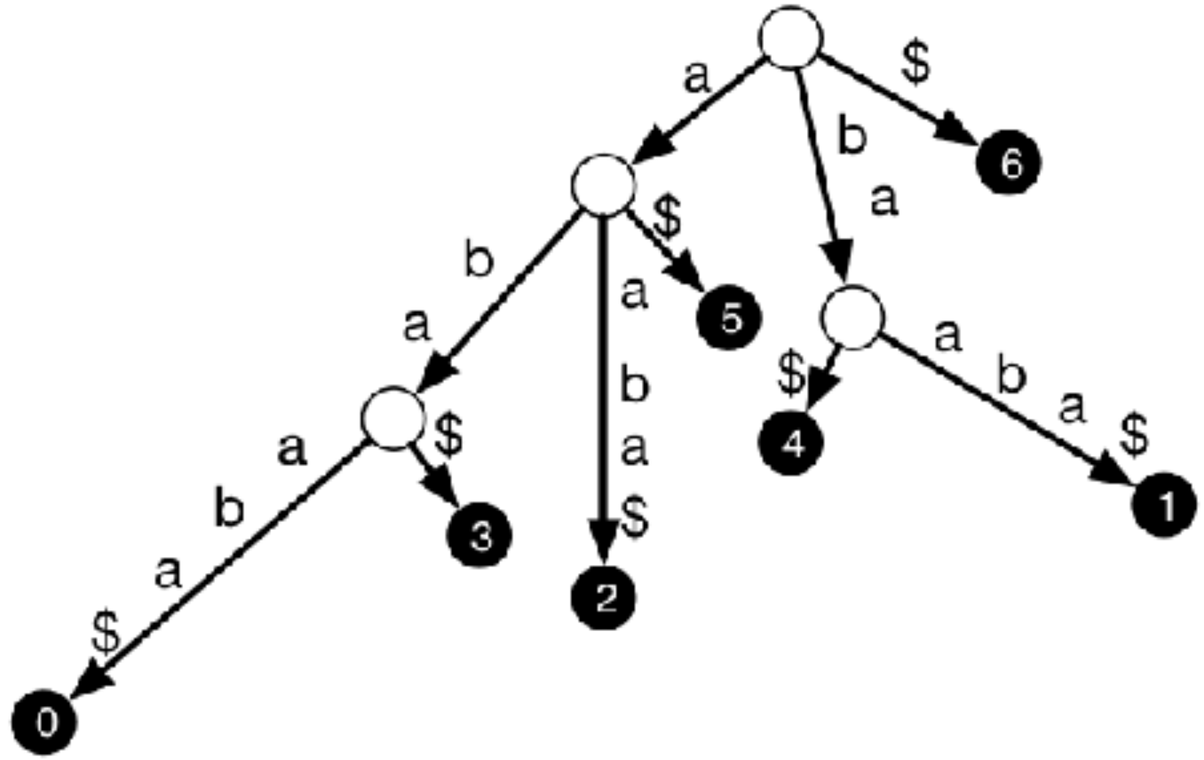
- Given:
  - 100’s of millions of short reads: 100-200bp reads
  - A long reference genome (~3Bbp for human)
- Do:
  - Find high scoring scoring (fitting) alignments for each read
- What we know:
  - Dynamic programming solution for fitting alignment:
    - 1e8 * 1e9 * 1e2 operations, 1e9 * 1e2 memory

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

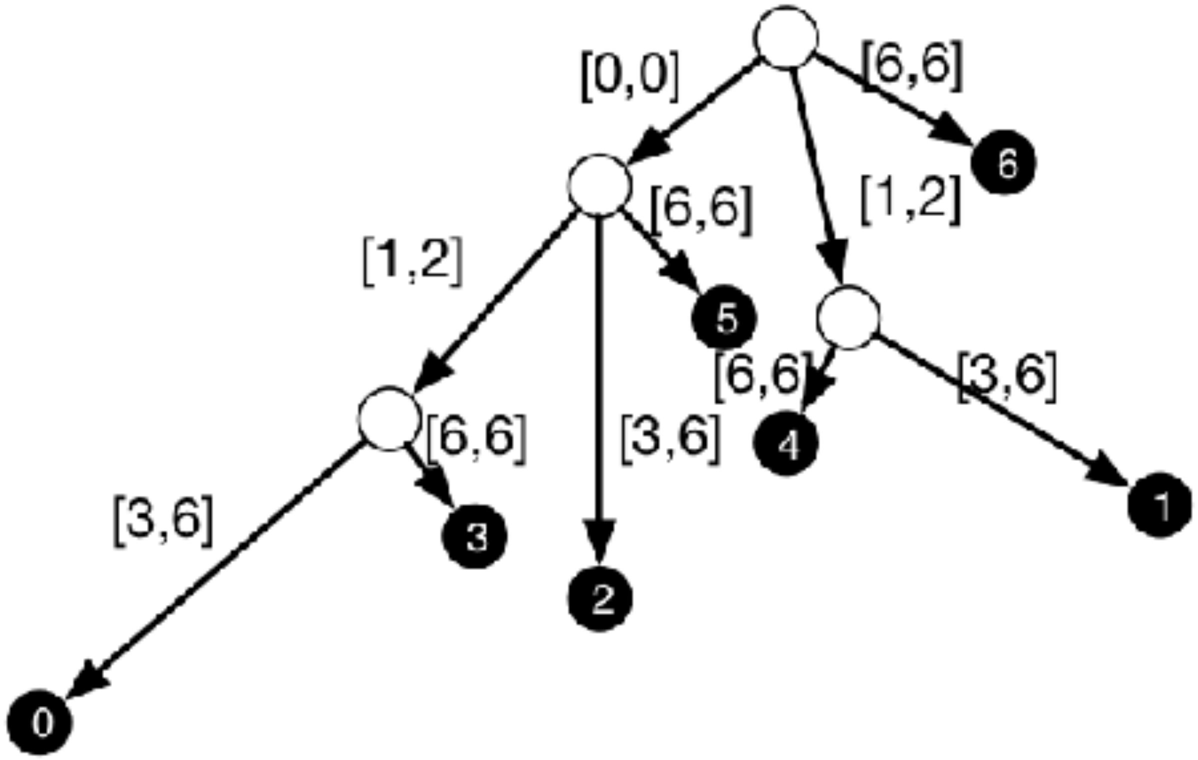
```
T: abaaba$
   0123456
```



- 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.

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

sort the suffixes alphabetically

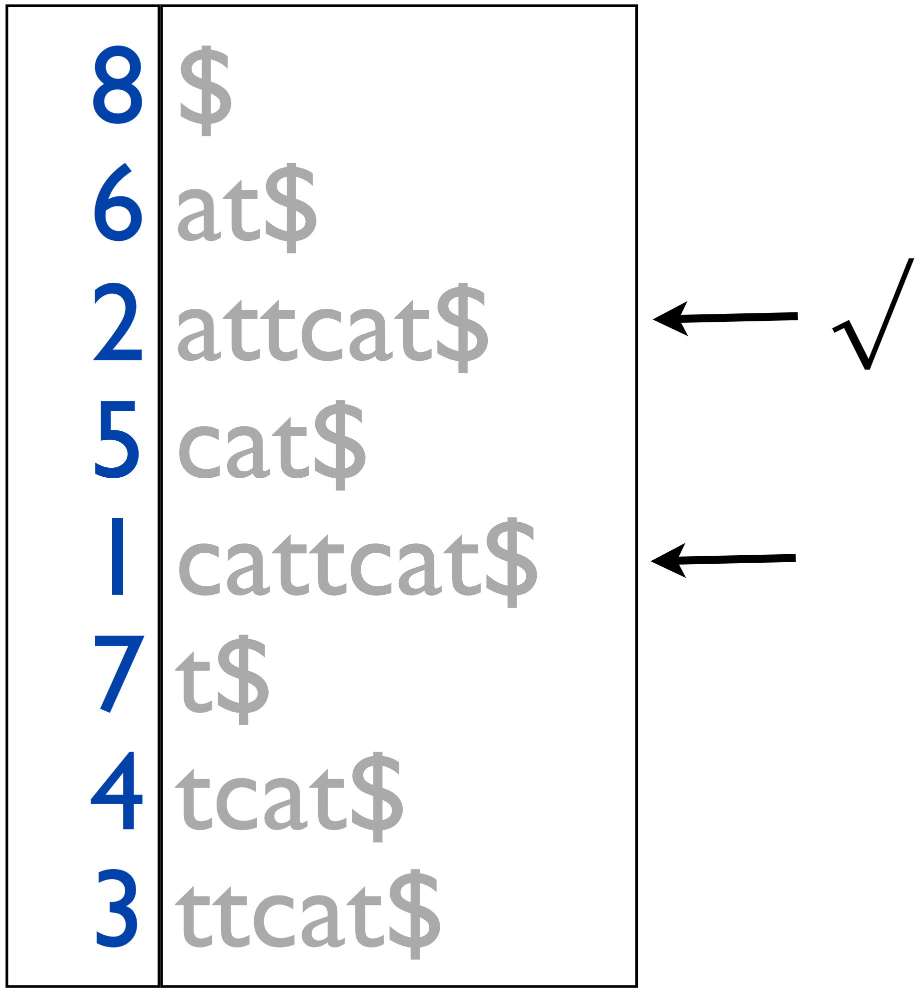
→

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:

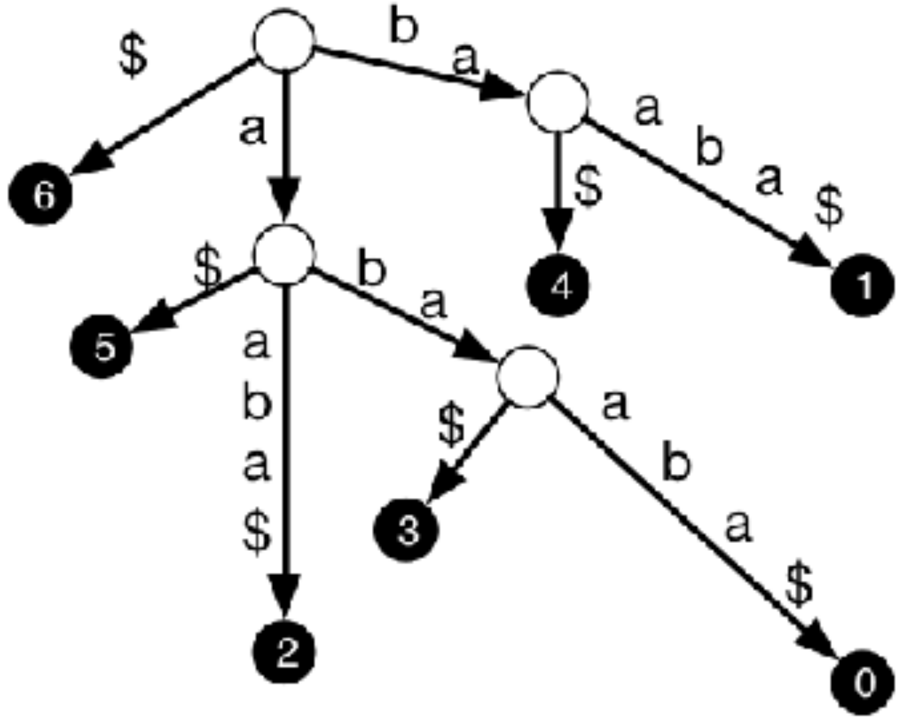
  sort the n suffixes, which takes O(n log n) comparisons, where each comparison takes O(n).

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

sort

$banana
a$banan
ana$ban
anana$b
banana$
nana$ba
na$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$
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

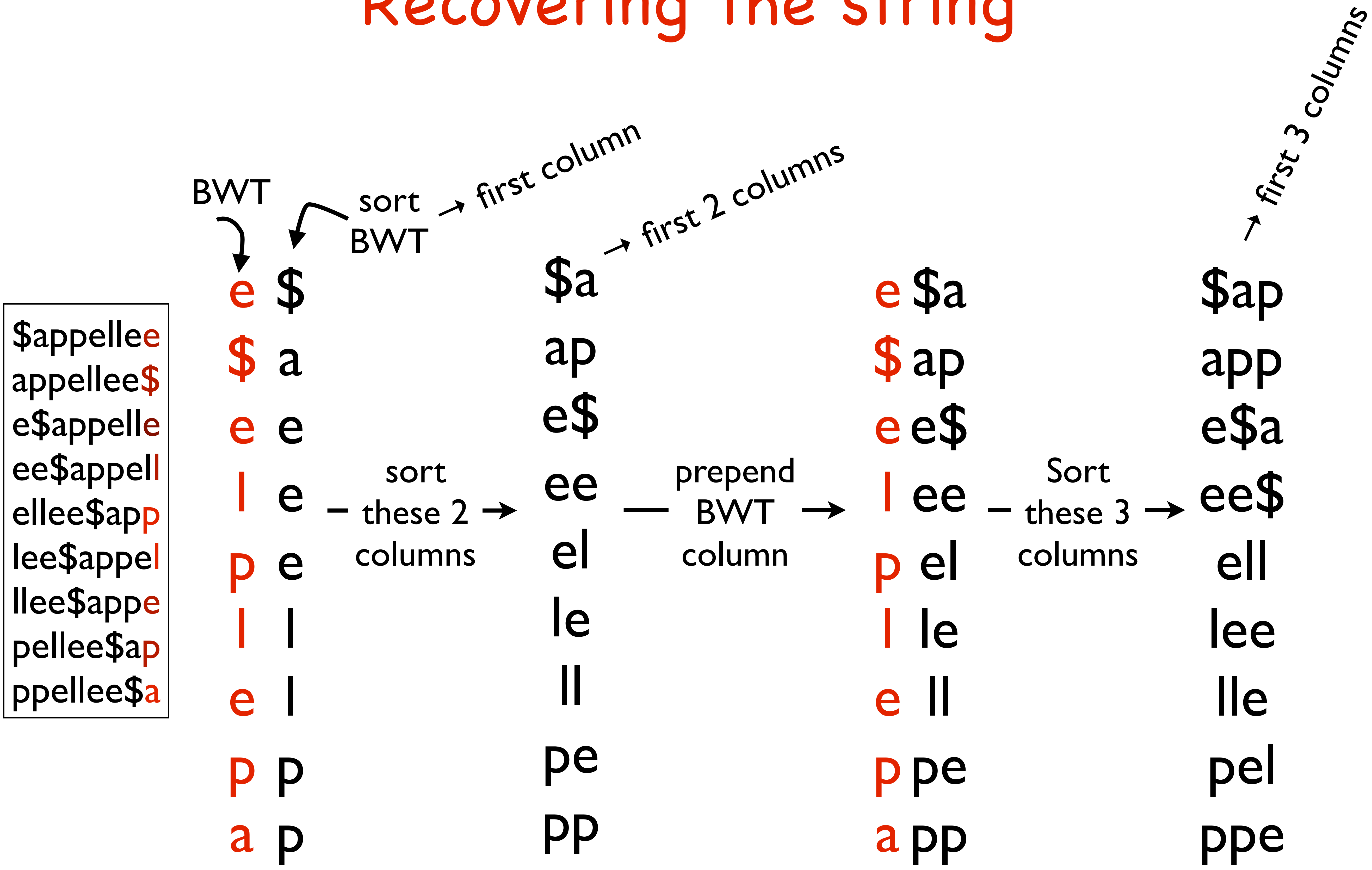
BWT(appellee$) =
e$elplepa

Doesn't always improve the compressibility...

1) Compute Burrows-Wheeler Transform of

abbaaba$

Recovering the string



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

BWT

sort BWT → first column

→ first 2 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

– sort these 2 columns →

— prepend BWT column →

– Sort these 3 columns →

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 $
```

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$d	od$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

Searching with BWT: LF Mapping

LF Mapping

Σ

BWT(unabashable)	$	a	b	e	h	l	n	s	u
$unabashable	0	0	0	0	0	0	0	0	0
abashable$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 ith occurrence of a letter X in the last column corresponds to the ith occurrence of X in the first column.

1) Compute Burrows-Wheeler Transform of

abbaaba$

2) Write down the LF mapping matrix for the BWT of question 1

BWT Search

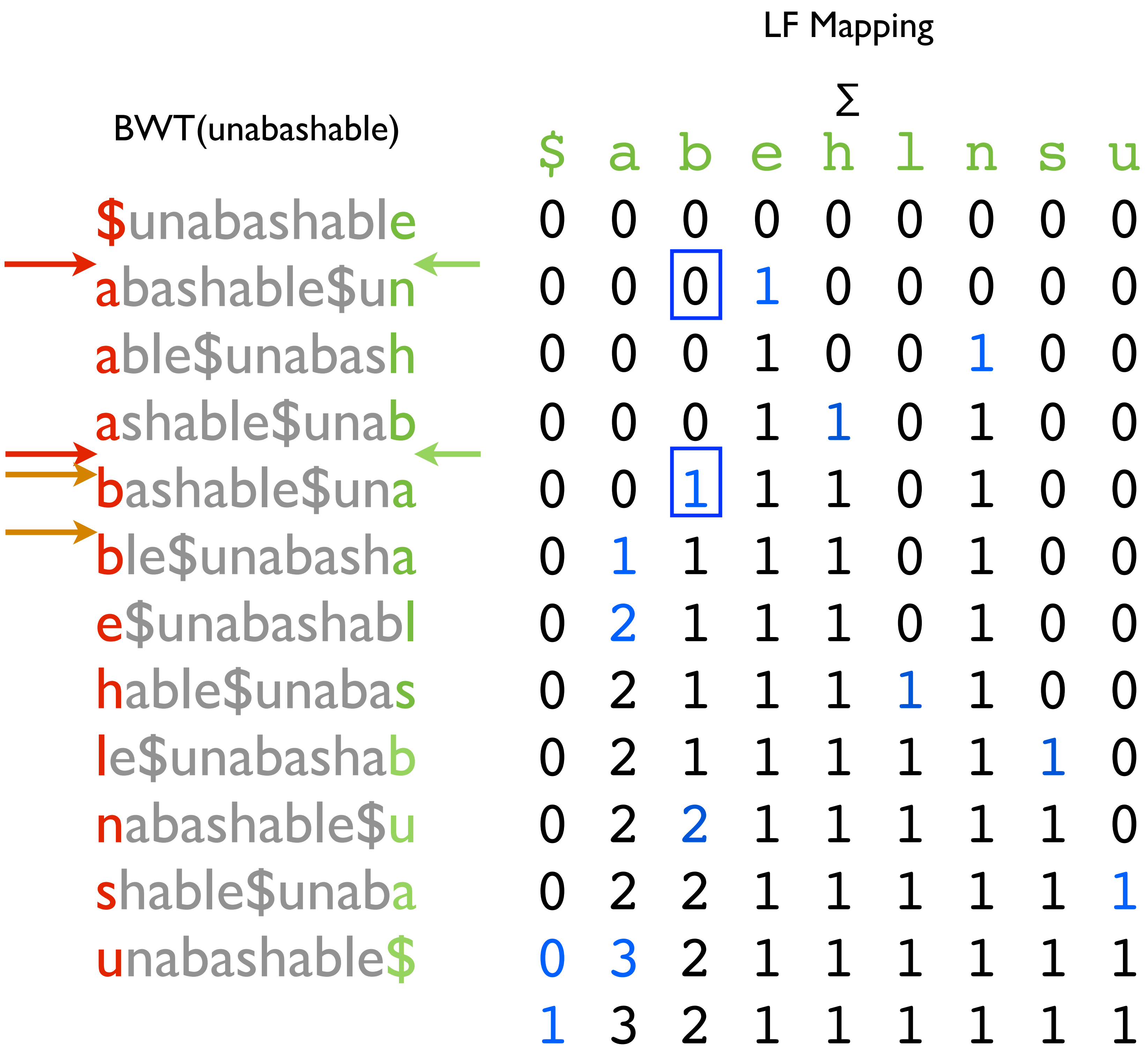
BWTSearch(aba) Start from the **end** of the pattern

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$unaba
unabashable$

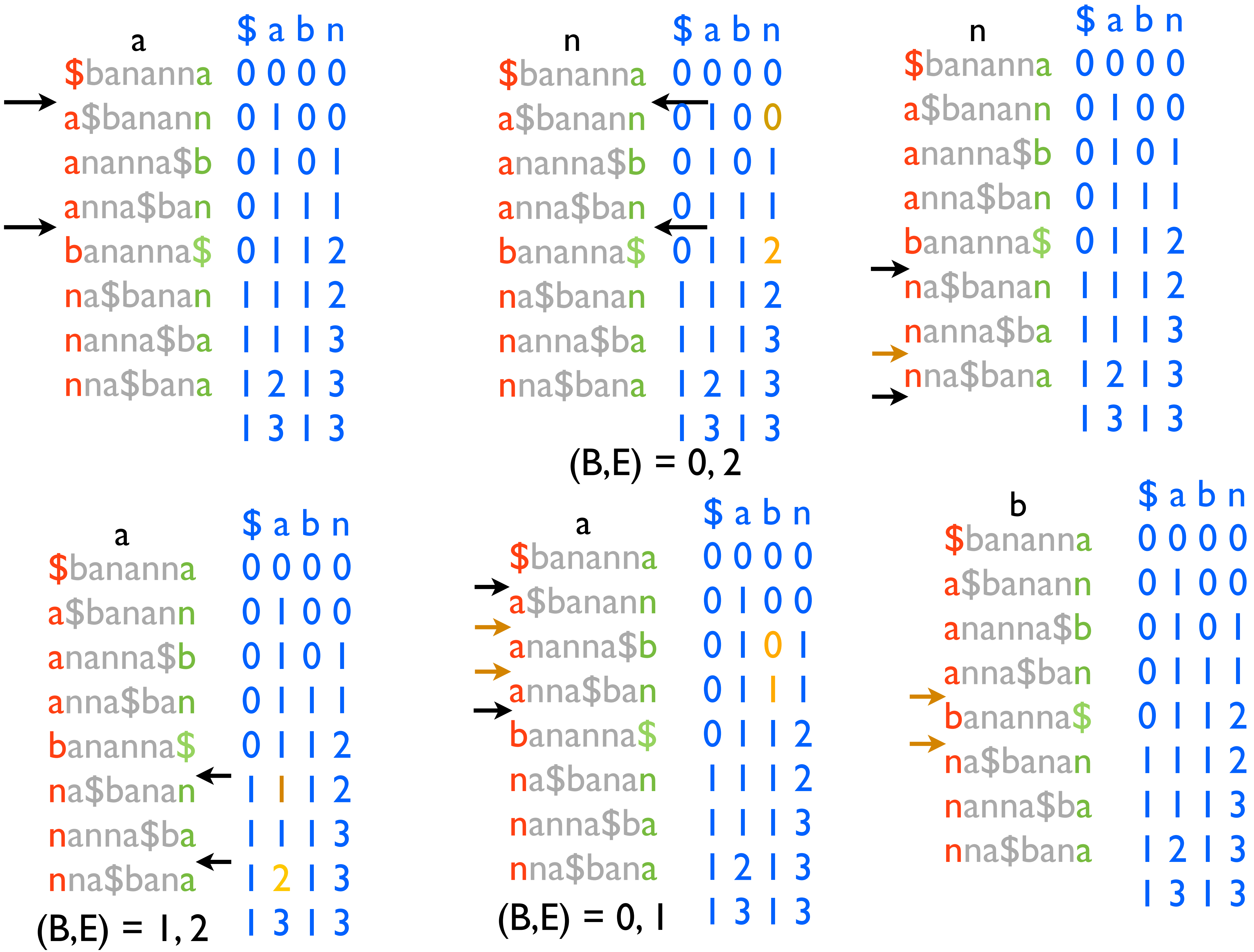
LF Mapping

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 (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.

Pseudocode for CountingOccurrences in BWT w/o stored LF mapping

```
function Count($S_{bwt}$, P):
    c = P[p], i = p
    sp = C[c] + 1; ep = C[c+1]

    while (sp ≤ ep) and (i ≥ 2) do
      c = P[i-1]
      sp = C[c] + Occ(c, sp-1) + 1
      ep = C[c] + Occ(c, ep)
      i = i - 1

    if ep < sp then
      return “not found”
    else
      return ep - sp + 1
```

C[c] = index into first column where the “c”’s begin.

Occ(*c*, *p*) = # of of *c* in the first *p* characters of BWT(S), aka the LF mapping.

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)

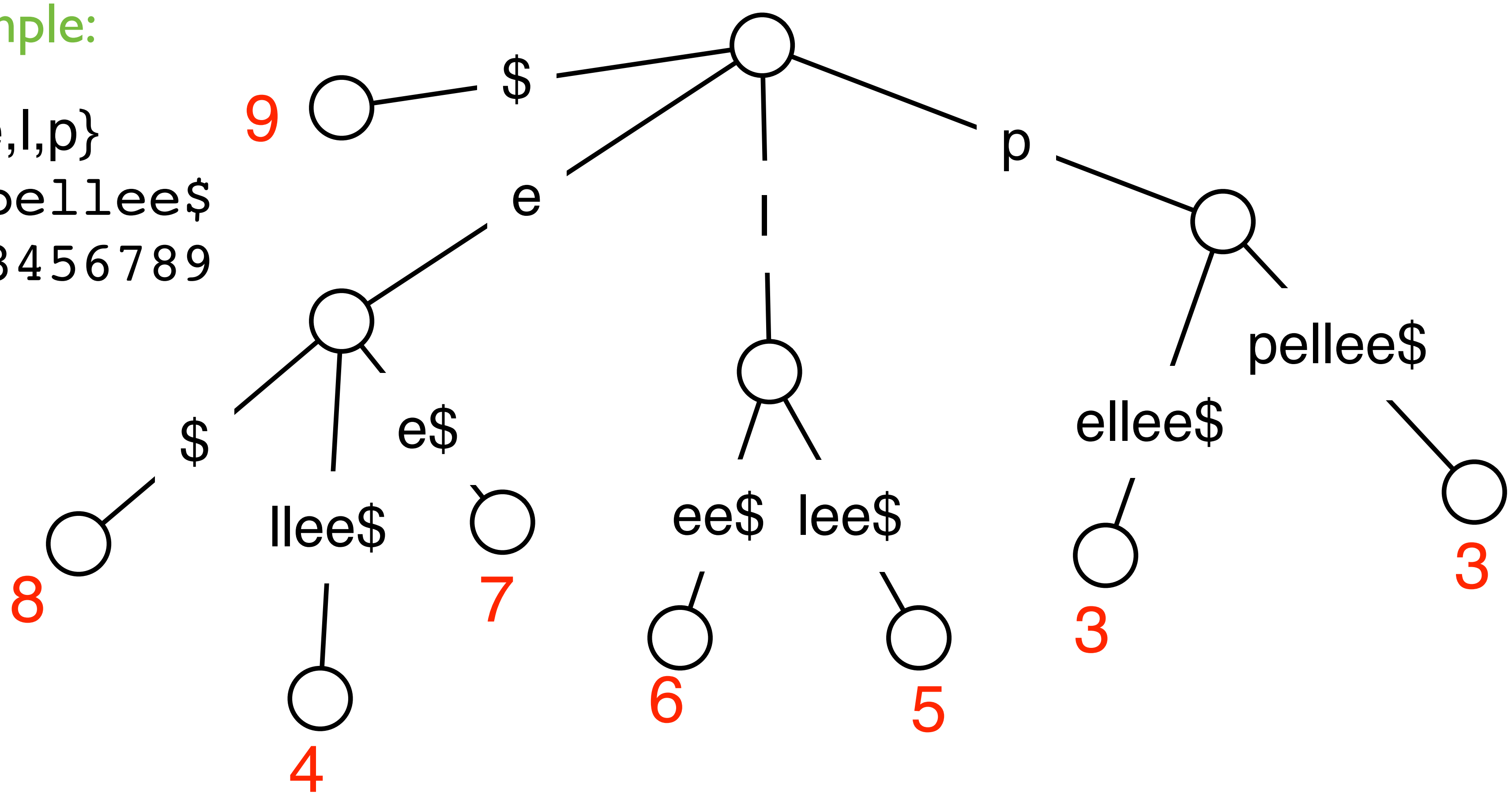
Relationship Between BWT and Suffix Trees

- 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:

Σ = {$,e,l,p}
s = `appellee$`
`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

Even after compression, can search string quickly.

Recap

Structure	Processing Time	Memory	Search
Suffix Trie	$O(\lvert T\rvert)$	$O(\lvert T\rvert^2)$	$O(\lvert P\rvert)$
Suffix Tree	$O(\lvert T\rvert)$	$O(\lvert T\rvert)$*	$O(\lvert P\rvert)$
Suffix Array	$O(\lvert T\rvert)$	$O(\lvert T\rvert)$ (but **much** smaller than Suffix Tree)	$O(\lvert P\rvert\log_2\lvert T\rvert)$
BWT	$O(\lvert T\rvert)$	$O(\lvert T\rvert)$**	$O(\lvert P\rvert)$

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

Move-To-Front Coding

To encode a letter, use its index in the current list, and then move it to the front of the list.

List with all letters from the allowed alphabet

Σ	do$oodwg
$dgow	1
d$gow	13
od$gw	132
$odgw	1322
o$dgw	13220
o$dgw	132202
do$gw	1322024
wdo$g	13220244 = MTF(do$oodwg)

Benefits:

- Runs of the same letter will lead to runs of 0s.
- Common letters get small numbers, while rare letters get big numbers.

Compressing BWT Strings

Lots of possible compression schemes will benefit from preprocessing with BWT (since it tends to group runs of the same letters together).

One good scheme proposed by Ferragina & Manzini:

replace runs of 0s
with the count of 0s

PrefixCode(rle(MTF(BWT(S))))

Huffman code that
uses more bits for
rare symbols

Pseudocode for CountingOccurrences in BWT w/o stored LF mapping

```
function Count($S_{bwt}$, P):

    c = P[p], i = p
    sp = C[c] + 1; ep = C[c+1]

    while (sp ≤ ep) and (i ≥ 2) do
      c = P[i-1]
      sp = C[c] + Occ(c, sp-1) + 1
      ep = C[c] + Occ(c, ep)
      i = i - 1

    if ep < sp then
      return “not found”
    else
      return ep - sp + 1
```

C[c] = index into first column where the “c”’s begin.

Occ(*c*, *p*) = # of of *c* in the first *p* characters of BWT(S), aka the LF mapping.

Computing Occ in Compressed String

Break BWT(S) into blocks of length L (we will decide on a value for L later):

BWT(S)

BT$_1$ | BT$_2$ | BT$_3$ | ...

PrefixCode(rle(MTF(BWT(BT$_2$))))

Occ(c, p) = # of "c" up thru p

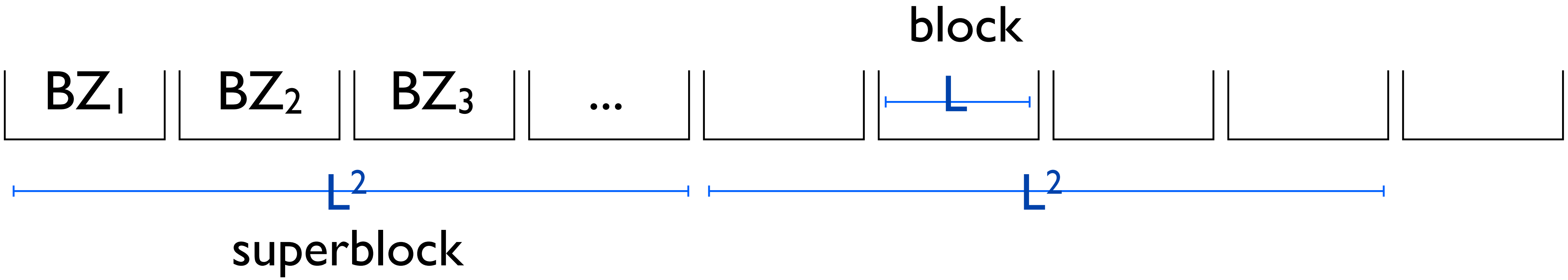
BZ$_1$ | BZ$_2$ | BZ$_3$ | ...

Assumes every run of 0s is contained in a block [just for ease of explanation].

We will store some extra info for each block (and some groups of blocks) to compute Occ(c, p) quickly.

Extra Info to Compute Occ

block: store |∑|-long array giving # of occurrences of each character up thru and including this block *since the end of the last super block.*



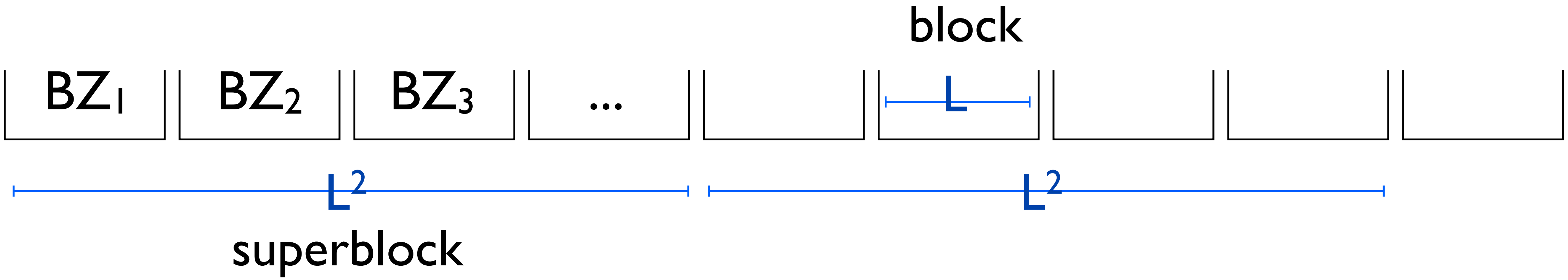
superblock: store |∑|-long array giving # of occurrences of each character up thru *and including* this superblock

Extra Info to Compute Occ

u = compressed length
Choose $L = O(\log u)$

u/L blocks, each array is $|\Sigma| \log L$ long ⇒
$\frac{u}{L} \log L = \frac{u}{\log u} \log \log u$ total space.

block: store $|\Sigma|$-long array giving # of occurrences of each character up thru and including this block *since the end of the last super block.*



block

BZ_1 BZ_2 BZ_3 …

L

L^2 L^2

superblock

superblock: store $|\Sigma|$-long array giving # of occurrences of each character up thru *and including* this superblock

Extra Info to Compute Occ

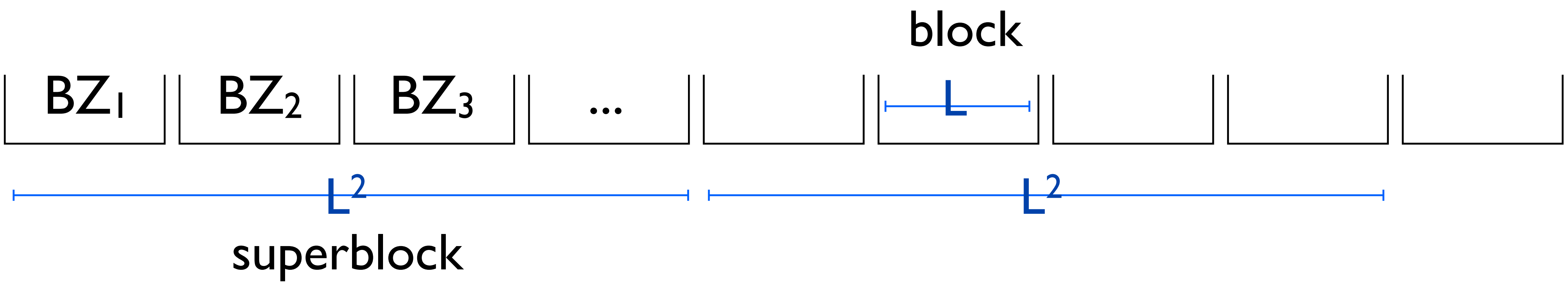
u = compressed length

Choose *L* = O(log *u*)

u/L blocks, each array is |∑|log *L* long ⇒

$\frac{u}{L} \log L = \frac{u}{\log u} \log \log u$ total space.

block: store |∑|-long array giving # of occurrences of each character up thru and including this block *since the end of the last super block.*



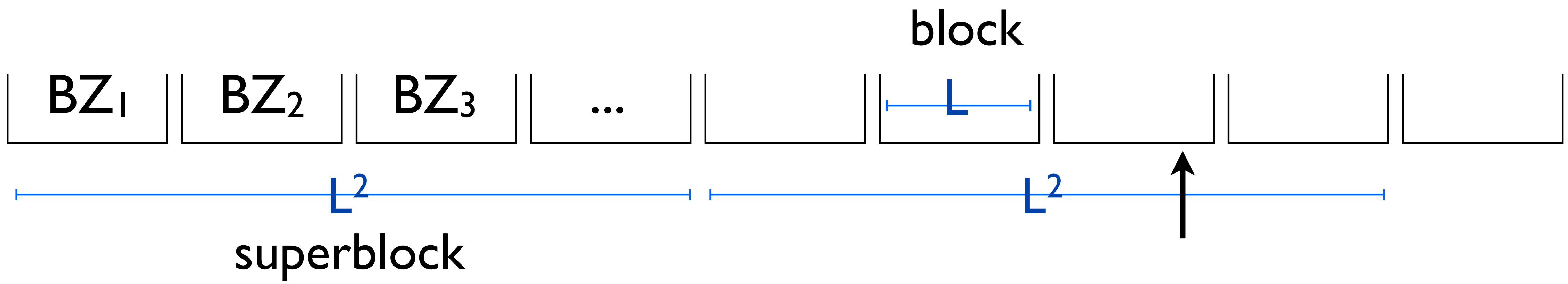
superblock: store |∑|-long array giving # of occurrences of each character up thru *and including* this superblock

u/L^2 superblocks, each array is |∑|log *u* long

$\Rightarrow \frac{u}{(\log u)^2} \log u = \frac{u}{\log u}$ total space.

Extra Info to Compute Occ

u = compressed length
Choose *L* = O(log *u*)



Occ(c, p) = # of "c" up thru p:

sum value at last superblock, value at end of previous block, but then need to handle *this block.*

Store an array: $M[c, k, BZ_i, MTF_i]$ = # of occurrences of c through the kth letter of a block of <u>type</u> (BZ_i, MTF_i).

Size: $O(|\Sigma| L 2^{L} |\Sigma|) = O(L 2^{L'}) = O(u^c \log u)$ for $c < 1$ (since the string is compressed)