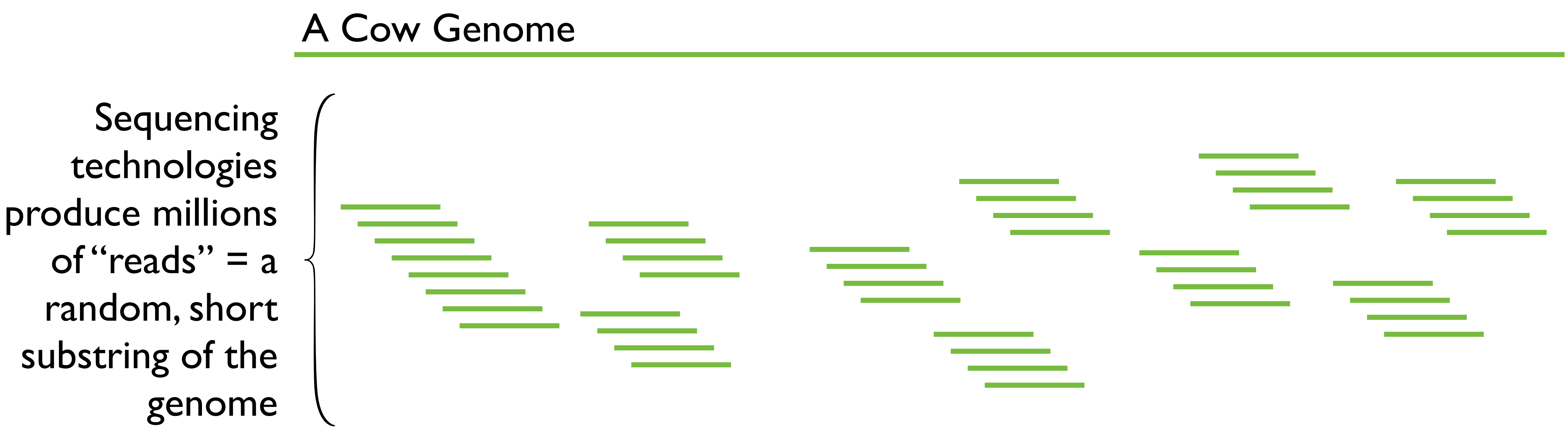
Burrows-Wheeler Transform

CMSC 423

Motivation – Short Read Mapping



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.

Bowtie

Software

Highly accessed Open access

Ultrafast and memory-efficient alignment of short DNA sequences to the human genome

Ben Langmead*, **Cole Trapnell**, **Mihai Pop** and **Steven L Salzberg**

* Corresponding author: Ben Langmead langmead@cs.umd.edu

▾ Author Affiliations

Center for Bioinformatics and Computational Biology, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA

For all author emails, please log on.

Genome Biology 2009, **10**:R25 doi:10.1186/gb-2009-10-3-r25

The electronic version of this article is the complete one and can be found online at: http://genomebiology.com/2009/10/3/R25

BWA

Fast and accurate short read alignment with Burrows–Wheeler transform

Heng Li and Richard Durbin*

+ Author Affiliations

* To whom correspondence should be addressed.

Received February 20, 2009.
Revision received May 6, 2009.
Accepted May 12, 2009.

Bioinformatics (2009) 25(14):1754-1760.

Bowtie Performance

Varying read length using Bowtie, Maq and SOAP

Length	Program	CPU time	Wall clock time	Peak virtual memory footprint (megabytes)	Bowtie speed-up	Reads aligned (%)
36 bp	Bowtie	6 m 15 s	6 m 21 s	1,305	-	62.2
	Maq	3 h 52 m 26 s	3 h 52 m 54 s	804	36.7×	65.0
	Bowtie -v 2	4 m 55 s	5 m 00 s	1,138	-	55.0
	SOAP	16 h 44 m 3 s	18 h 1 m 38 s	13,619	216×	55.1
50 bp	Bowtie	7 m 11 s	7 m 20 s	1,310	-	67.5
	Maq	2 h 39 m 56 s	2 h 40 m 9 s	804	21.8×	67.9
	Bowtie -v 2	5 m 32 s	5 m 46 s	1,138	-	56.2
	SOAP	48 h 42 m 4 s	66 h 26 m 53 s	13,619	691×	56.2
76 bp	Bowtie	18 m 58 s	19 m 6 s	1,323	-	44.5
	Maq 0.7.1	4 h 45 m 7 s	4 h 45 m 17 s	1,155	14.9×	44.9
	Bowtie -v 2	7 m 35 s	7 m 40 s	1,138	-	31.7

Maq & SOAP build hash table of locations of k-mers

The performance of Bowtie v0.9.6, SOAP v1.10, and Maq versions v0.6.6 and v0.7.1 on the server platform when aligning 2 M untrimmed reads from the 1,000 Genome project (National Center for Biotechnology Information Short Read Archive: SRR003084 for 36 base pairs [bp], SRR003092 for 50 bp, and SRR003196 for 76 bp). For each read length, the 2 M reads were randomly sampled from the FASTQ file downloaded from the Archive such that the average per-base error rate as measured by quality values was uniform across the three sets. All reads pass through Maq's "catfilter". Maq v0.7.1 was used for the 76-bp reads because v0.6.6 does not support reads longer than 63 bp. SOAP is excluded from the 76-bp experiment because it does not support reads longer than 60 bp. Other experimental parameters are identical to those of the experiments in Table 1. CPU, central processing unit.

Langmead et al. (2008)

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

$appellee
appellee$
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

BWT	first column	first 2 columns	prepend BWT column	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 = [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$d	od$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
$unabashable	0	0	0	0	0	0	0	0	0
ababashable$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.

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	$\boxed{0}$	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	$\boxed{1}$	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
```

$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	− 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

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

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)

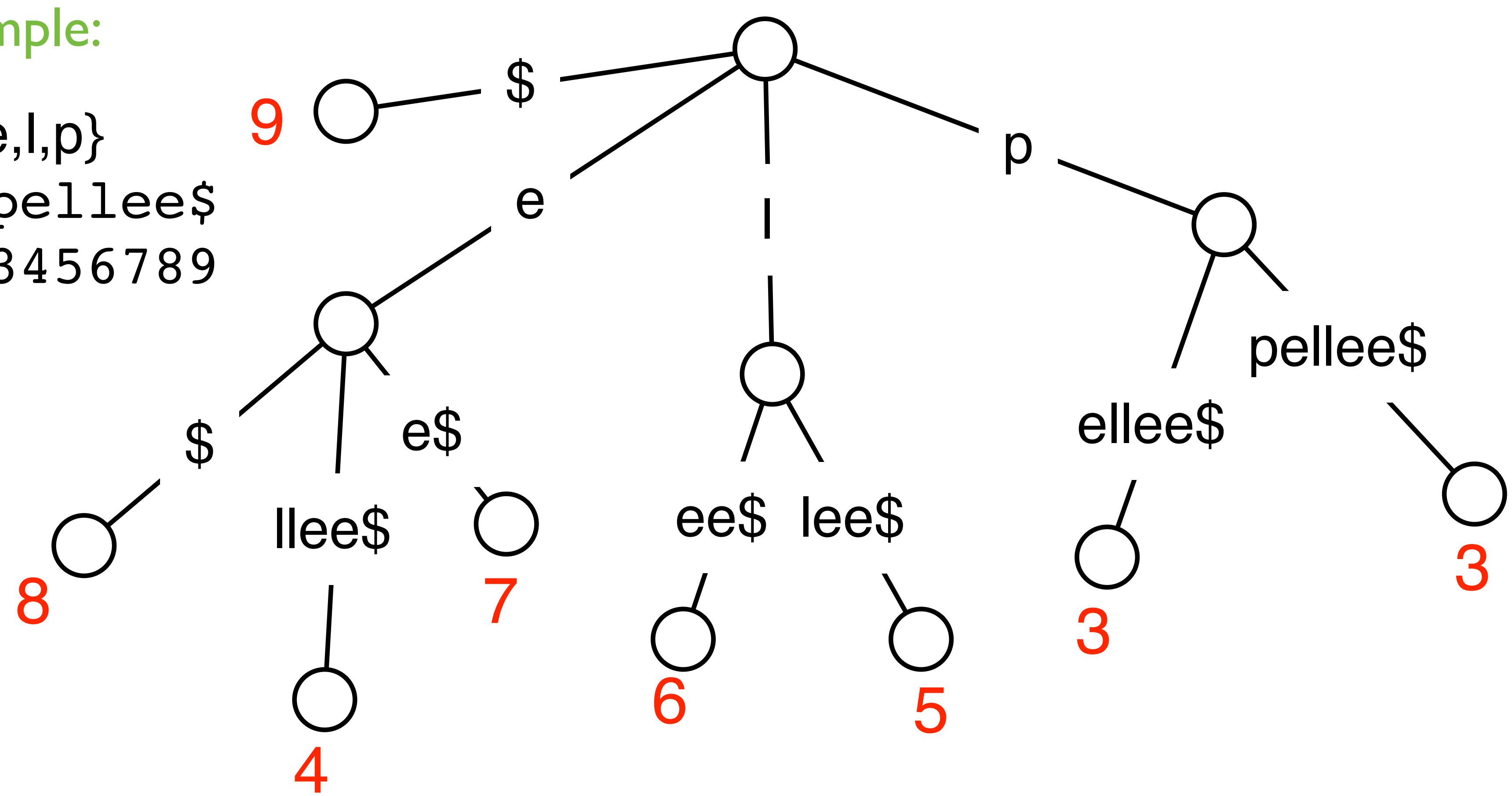
Relationship Between BWT and Suffix Trees

- 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.

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 <u>count</u> 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

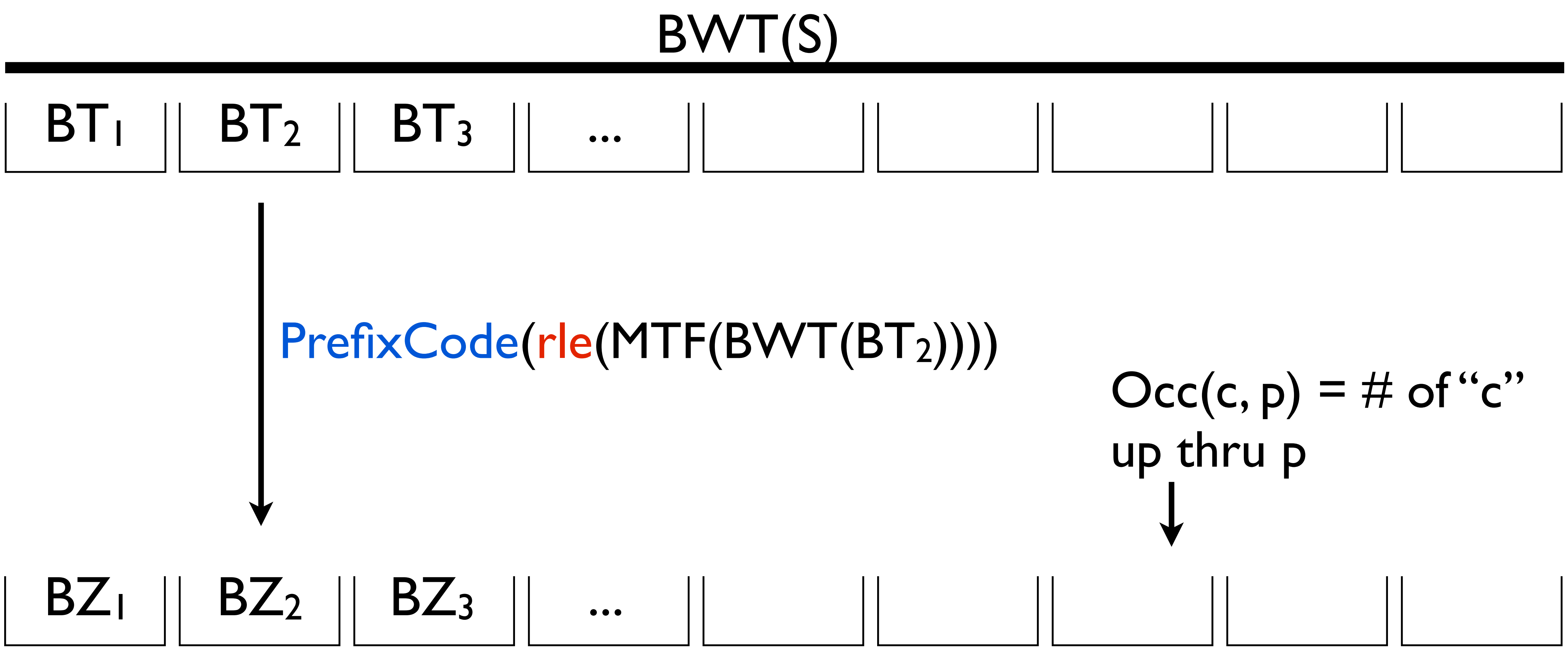
```
function Count(Sbwt, P):

    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):

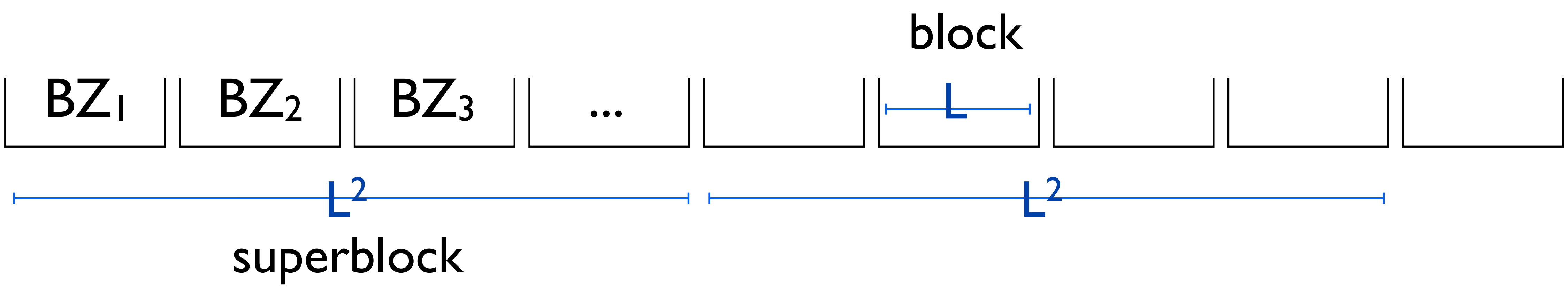


Assumes every run of 0s is contained in a block [just for ease of explanation].

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.*



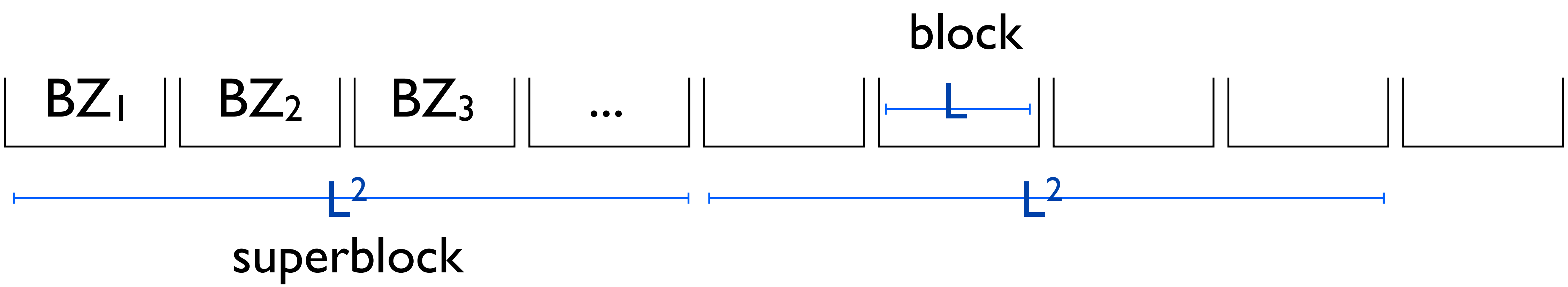
superblock: store |∑|-long array giving # of occurrences of each character up thru *and including* this superblock

Extra Info to Compute Occ

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

u/L blocks, each array is $|\Sigma| \log L$ long $\Rightarrow$
$\frac{u}{L} \log L = \frac{u}{\log u} \log \log u$ 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.*



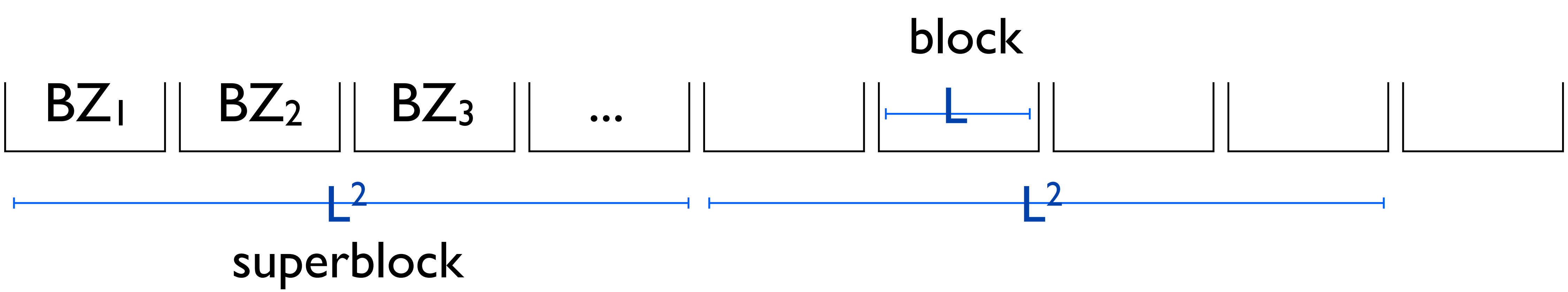
superblock: store $|\Sigma|$-long array giving # of occurrences of each character up thru *and including* this superblock

Extra Info to Compute Occ

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

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.*

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

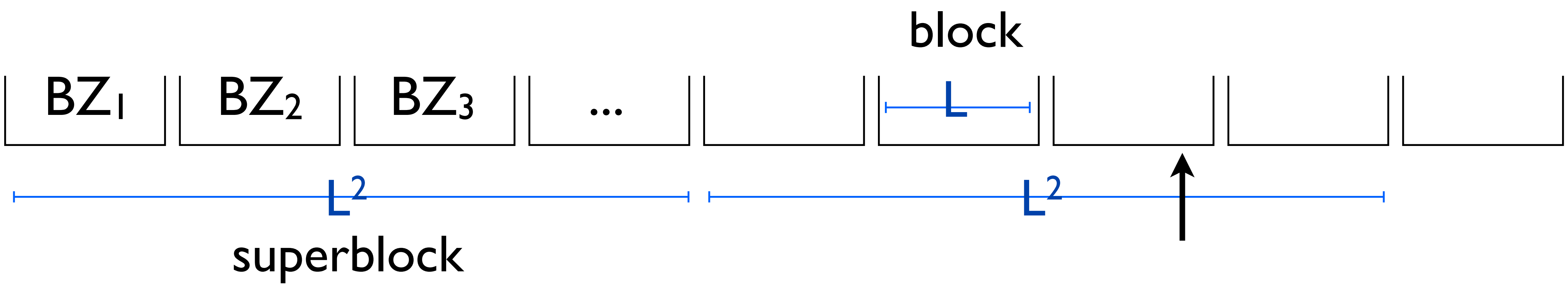


superblock: store $|\Sigma|$-long array giving # of occurrences of each character up thru *and including* this superblock

u/L^2 superblocks, each array is $|\Sigma| \log u$ 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)

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.