The Burrows-Wheeler Transform and Bioinformatics

J. Matthew Holt
holtjma@cs.unc.edu

Last Class - Multiple Pattern Matching Problem

m - length of text
d - max length of pattern
x - number of patterns

Method	Storage Cost	Single Pattern Search Time	Multiple Pattern Search Time
Brute Force	O(m)	O(dm)	O(xdm)
Keyword Tries	O(xd)	O(dm)	O(dm)
Suffix Trees	O(m) [20m bytes]	O(d)	O(xd)
Suffix Arrays	O(m*log(m)) [4m bytes]	O(d*log(m))	O(xd*log(m))
BWT	**O(m) [often m bits]**	**O(d)**	**O(xd)**

Recall Suffix Arrays

- Create all suffix - note we are doing **cyclic** suffixes now
- Sort the suffixes
- Indices are stored

Index (*N*)	Rotations	Suffix Array	Sorted Suffixes
0	ACACGGACA$	9	$ACACGGACA
1	CACGGACA$A	8	A$ACACGGAC
2	ACGGACA$AC	6	ACA$ACACGG
3	CGGACA$ACA	0	ACACGGACA$
4	GGACA$ACAC	2	ACGGACA$AC
5	GACA$ACACG	7	CA$ACACGGA
6	ACA$ACACGG	1	CACGGACA$A
7	CA$ACACGGA	3	CGGACA$ACA
8	A$ACACGGAC	5	GACA$ACACG
9	$ACACGGACA	4	GGACA$ACAC

The suffix array for string "ACACGGACA$"
"$" is just an end-of-string character

Recall Suffix Arrays (cont.)

- ***N*** - number of bases (length of text)
- ***k*** - pattern length (also called a ***k*-mer**)
- Space complexity: **O(*N**log(*N*))** bits
  - Stored as offsets into original string
  - N offsets that require log(N) bits per value
- Search time: **O(*k**log(*N*))** operations
  - Binary search require O(log(*N*)) string comparisons
  - Each string comparison requires O(*k*) symbol comparisons
- Problem:
  - O(*k**log(*N*)) can be large when strings are billions of characters long

The Burrows-Wheeler Transform

- Burrows & Wheeler, 1994
- **BWT** - a permutation of a string that implicitly represent a suffix array of the same string
- Naive construction:
  - Create all suffixes
  - Sort suffixes
  - Concatenate last symbols in suffixes
- Implicit suffix array
  - "Last symbol" in suffix
  - "Previous symbol" to suffix

Index (N)	Rotations	Sorted Suffixes	BWT
0	ACACGGACA$	$ACACGGAC**A**	**A**
1	CACGGACA$A	A$ACACGGA**C**	**C**
2	ACGGACA$AC	ACA$ACACG**G**	**G**
3	CGGACA$ACA	ACACGGACA**$**	**$**
4	GGACA$ACAC	ACGGACA$A**C**	**C**
5	GACA$ACACG	CA$ACACGG**A**	**A**
6	ACA$ACACGG	CACGGACA$**A**	**A**
7	CA$ACACGGA	CGGACA$AC**A**	**A**
8	A$ACACGGAC	GACA$ACAC**G**	**G**
9	$ACACGGACA	GGACA$ACA**C**	**C**

BWT algorithm

BWT (string text)

$table_i$ = Rotate(text, i) for i = 0..len(text)-1

sort table alphabetically

return (last column of the table)

```
tarheel$        $tarheel
arheel$t        arheel$t
rheel$ta        eel$tarh
heel$tar        el$tarhe
eel$tarh        heel$tar
el$tarhe        l$tarhee
l$tarhee        rheel$ta
$tarheel        tarheel$
```

BWT(“tarheels$”) = “ltherea$”

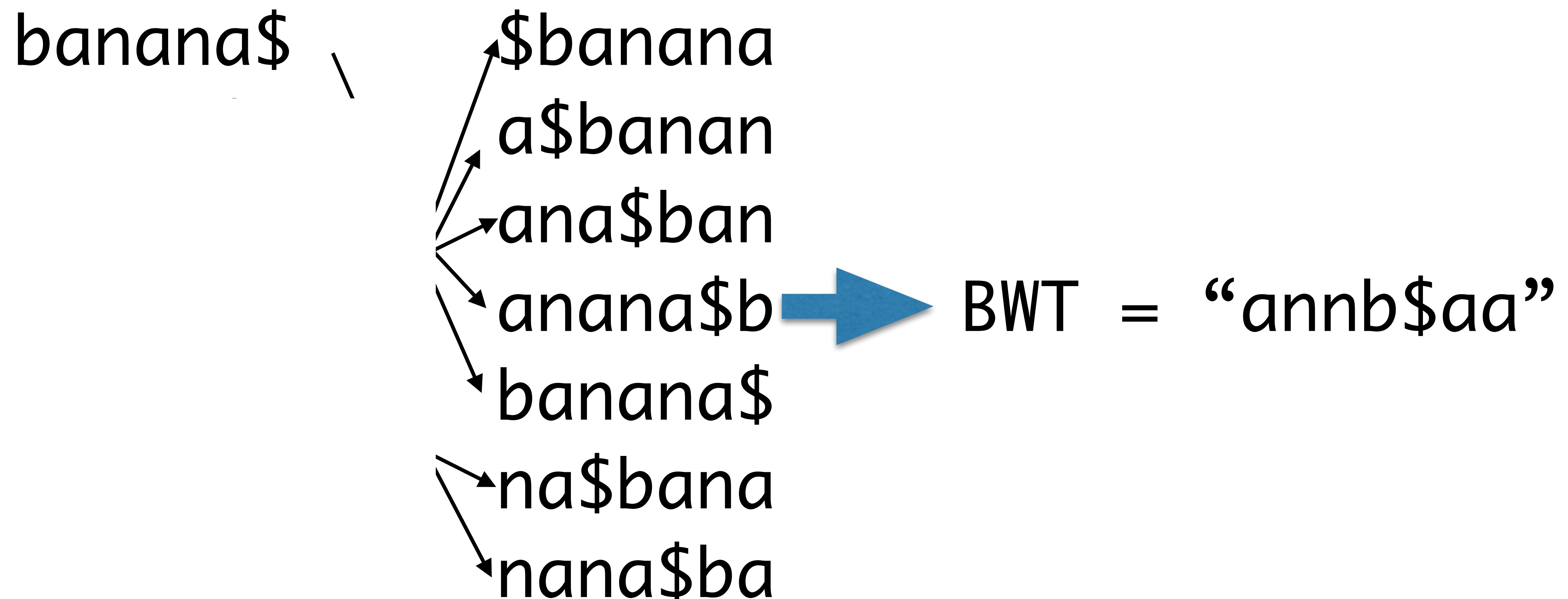
Example

BWT (string text)

table$_i$ = Rotate(text, i) for i = 0..len(text)-1

sort table alphabetically

return (last column of the table)



BWT in Python

t - a text that we want a BWT of

```
def BWT(t):
  #create a list of all cyclic suffixes of t
  rotation = [t[i:]+t[:i] for i in xrange(len(t))]
  #sort the suffixes
  rotation.sort()
  #concatenate the last symbol from each suffix
  return “”.join(r[-1] for r in rotation)
```

sa - the suffix array for text *t* (see last slides)

```
def BWT_fromSA(t, sa):
  return “”.join(t[v-1] for v in sa)
```

Inverting a BWT

- A property of a transform is that there is no information loss and they are invertible.

```
inverseBWT(string s)
    add s as the first column of a table strings
    repeat length(s)-1 times:
        sort rows of the table alphabetically
        add s as the first column of the table
    return (row that ends with the 'EOF' character)
```

```
l   l$   l$t   l$ta   l$tar   l$tarh   l$tarhe   l$tarhee
t   ta   tar   tarh   tarhe   tarhee   tarheel   tarheel$
h   he   hee   heel   heel$   heel$t   heel$ta   heel$tar
e   ee   eel   eel$   eel$t   eel$ta   eel$tar   eel$tarh
r   rh   rhe   rhee   rheel   rheel$   rheel$t   rheel$ta
e   el   el$   el$t   el$ta   el$tar   el$tarh   el$tarhe
a   ar   arh   arhe   arhee   arheel   arheel$   arheel$t
$   $t   $ta   $tar   $tarh   $tarhe   $tarhee   $tarheel
```

Inverting in Python

```
def inverseBWT(t):
  #initialize the table from t
  table = [c for c in t]
  for j in xrange(len(t)-1):
    #sort the table
    table.sort()
    #insert the BWT as the first column
    table = [t[i]+table[i] for i in xrange(len(t))]
  #return the row that ends with end-of-string ‘$’
  return table[[r[-1] for r in table].index(‘$’)]
```

BWT Compression

- Uncompressed
  - Same as input text size
  - O(*N*) bytes
- Compression
  - Tendency to form long runs
  - Run-length encoding (RLE)
- Can be stored as: ACG$C3AGC

Index (N)	Suffix Array	BWT
0	$ACACGGAC**A**	**A**
1	A$ACACGGA**C**	**C**
2	ACA$ACACG**G**	**G**
3	ACACGGACA**$**	**$**
4	ACGGACA$A**C**	**C**
5	CA$ACACGG**A**	**A**
6	CACGGACA$**A**	**A**
7	CGGACA$AC**A**	**A**
8	GACA$ACAC**G**	**G**
9	GGACA$ACA**C**	**C**

Why does it form runs?

- Think about a BWT and suffix array of a book such as "Green Eggs and Ham"

Sorted Suffixes	BWT
…	**…**
ould you eat them in a box? …	**w**
ould you eat them with a fox? …	**w**
ould you like them here or there? …	**w**
ould you like them in a house? …	**w**
ould you like them with a mouse? …	**w**
…	**…**

- Assumption is that patterns that cluster together will have the same symbol preceding them; thus the BWT forms long runs

FM-Index

- Ferragina & Manzini, 2005
- Enables fast exact searches
- **LF-mapping property** - Takes advantage of "last-first" relationship between BWT and suffix array
  - See colors on right
  - First "**A**" in BWT corresponds to first suffix starting with "**A**"
  - First "**C**" in BWT corresponds to first suffix starting with "**C**"
  - …
  - Second "**A**" in BWT corresponds to second suffix starting with "**A**"
  - …

Index (N)	Suffix Array	BWT
0	$ACACGGACA	A
1	A$ACACGGAC	C
2	ACA$ACACGG	G
3	ACACGGACA$	$
4	ACGGACA$AC	C
5	CA$ACACGGA	A
6	CACGGACA$A	A
7	CGGACA$ACA	A
8	GACA$ACACG	G
9	GGACA$ACAC	C

FM-index (cont.)

- ***A*** - alphabet size
- ***N*** - text length
- ***F*** - ***FM-index***
  - *F*[*i*][*c*] stores the number of times symbol *c* occurs before index *i*
  - O(*NA*) memory
  - Generated in a linear pass over the BWT
- ***O*** - Offset Array
  - *O*[*c*] stores the index of the first suffix starting with symbol *c*
  - Derived from the final entry in *F* `O[c] = sum(F[-1][0:c])`
  - O(*A*) memory

Index (N)	Suffix Array (not stored)	BWT	FM-index (F)			
			$	A	C	G
0	$ACACGGAC**A**	**A**	0	0	0	0
1	A$ACACGGA**C**	**C**	0	1	0	0
2	ACA$ACACG**G**	**G**	0	1	1	0
3	ACACGGACA**$**	**$**	0	1	1	1
4	ACGGACA$A**C**	**C**	1	1	1	1
5	CA$ACACGG**A**	**A**	1	1	2	1
6	CACGGACA$**A**	**A**	1	2	2	1
7	CGGACA$AC**A**	**A**	1	3	2	1
8	GACA$ACAC**G**	**G**	1	4	2	1
9	GGACA$ACA**C**	**C**	1	4	2	2
10	—	**–**	**1**	**4**	**3**	**2**
Offset (O)	—	**–**	**0**	**1**	**5**	**8**

Find Predecessor Suffix

- Given an index *i* in the BWT, find the index in the BWT of the suffix preceding the suffix represented by *i*
  - suffix 0 is preceded by suffix 1
  - suffix 1 is preceded by suffix 5
  - suffix 5 is preceded by suffix 2
- The predecessor suffix of index *i:*

```
c = BWT[i]
predec = O[c]+F[i][c]
```

- Predecessor of index 1
  c = BWT[1] = 'C'
  predec = O['C']+F[1]['C'] = 5+0 = 5
- Predecessor of index 8
  c = BWT[8] = 'G'
  predec = O['G']+F[8]['G'] = 8+1 = 9
- Time to find predecessor: **O(1)**

Index (N)	Suffix Array (not stored)	BWT	FM-index (F)			
			$	A	C	G
0	$ACACGGAC**A**	**A**	0	0	0	0
1	A$ACACGGA**C**	**C**	0	1	0	0
2	ACA$ACACG**G**	**G**	0	1	1	0
3	ACACGGACA**$**	**$**	0	1	1	1
4	ACGGACA$A**C**	**C**	1	1	1	1
5	CA$ACACGG**A**	**A**	1	1	2	1
6	CACGGACA$**A**	**A**	1	2	2	1
7	CGGACA$AC**A**	**A**	1	3	2	1
8	GACA$ACAC**G**	**G**	1	4	2	1
9	GGACA$ACA**C**	**C**	1	4	2	2
10	—	**—**	**1**	**4**	**3**	**2**
Offset (O)	—	**—**	**0**	**1**	**5**	**8**

Suffix Recovery

- Suffix recovery:
  - Start at an index *i*
  - Repeatedly find the predecessor
  - Stop when back at original index
- Original string recovery:
  - Start at 0
  - Repeatedly find predecessor until back at 0
  - **O(*N*)** time to get original string back

```
def recoverSuffix(BWT, F, O, i):
  ret = []
  c = BWT[i]
  predec = O[c]+F[i][c]
  ret.append(c)
  while predec != i:
    c = BWT[predec]
    predec = O[c]+F[predec][c]
    ret.append(c)
  return “”.join(ret[::-1])
```

Find *k*-mer

- ***k*-mer**: a pattern of length *k*
- All searches occur in reverse order
  - Start with full BWT range (0, N)
  - Restrict by one symbol at a time
- Find *k*-mer “ACA”
  - Initialize to full range (“”)
    low, high = 0, 10
  - Find occurrences of “A”
    low = O[‘A’]+F[low][‘A’] = 1+0 = 1
    high = O[‘A’]+F[high][‘A’] = 1+4 = 5
  - Find occurrences of “CA”
    low = O[‘C’]+F[low][‘C’] = 5+0 = 5
    high = O[‘C’]+F[high][‘C’] = 5+2 = 7
  - Find occurrences of “ACA”
    low = O[‘A’]+F[low][‘A’] = 1+1 = 2
    high = O[‘A’]+F[high][‘A’] = 1+3 = 4

Index (N)	Suffix Array (not stored)	BWT	FM-index (F)			
			$	A	C	G
0	$ACACGGAC**A**	**A**	0	0	0	0
1	A$ACACGGA**C**	**C**	0	1	0	0
2	ACA$ACACG**G**	**G**	0	1	1	0
3	ACACGGACA**$**	**$**	0	1	1	1
4	ACGGACA$A**C**	**C**	1	1	1	1
5	CA$ACACGG**A**	**A**	1	1	2	1
6	CACGGACA$**A**	**A**	1	2	2	1
7	CGGACA$AC**A**	**A**	1	3	2	1
8	GACA$ACAC**G**	**G**	1	4	2	1
9	GGACA$ACA**C**	**C**	1	4	2	2
10	—	**—**	**1**	**4**	**3**	**2**
Offset (O)	—	**—**	**0**	**1**	**5**	**8**

Find *k*-mer (cont.)

- *p* - pattern
- *F* - FM-index
- Time complexity - O(*k*)
  - Requires O(*k*) lookups
  - Search time only dependent on length of *k*-mer
  - Does not depend on BWT (data) size!!!

```
def find(p, F, O):
  lo = 0
  hi = len(F)
  for l in reversed(p):
    lo = O[l] + F[lo][l]
    hi = O[l] + F[hi][l]
  return lo, hi
```

find("AGG", F, O)

```
def find(p, F, O):
  lo = 0
  hi = len(F)
  for l in reversed(p):
    lo = O[l] + F[lo][l]
    hi = O[l] + F[hi][l]
  return lo, hi
```

Index (N)	Suffix Array (not stored)	BWT	FM-index (F)			
			$	A	C	G
0	$ACACGGAC**A**	**A**	0	0	0	0
1	A$ACACGGA**C**	**C**	0	1	0	0
2	ACA$ACACG**G**	**G**	0	1	1	0
3	ACACGGACA**$**	**$**	0	1	1	1
4	ACGGACA$A**C**	**C**	1	1	1	1
5	CA$ACACGG**A**	**A**	1	1	2	1
6	CACGGACA$**A**	**A**	1	2	2	1
7	CGGACA$AC**A**	**A**	1	3	2	1
8	GACA$ACAC**G**	**G**	1	4	2	1
9	GGACA$ACA**C**	**C**	1	4	2	2
10	—	**—**	**1**	**4**	**3**	**2**
Offset (O)	—	**—**	**0**	**1**	**5**	**8**

Practical Adaptations

- Compressed BWT is small
- FM-index is not, O(*A*N*) for alphabet of size *A* and a BWT of length *N*
- Trade-off, space v. time:
  - Use a sampled FM-index
  - *B* - bin size
  - Uses O(*A*N*/*B*) values
  - Requires O(*B*) time per lookup (for a fixed size *B*, this is just a larger constant time lookup)
  - I typically use 1024 in practice

Index (N)	Suffix Array (not stored)	BWT	FM-index (F)			
			$	**A**	**C**	**G**
0	$ACACGGAC**A**	**A**	**0**	**0**	**0**	**0**
1	A$ACACGGA**C**	**C**	0	1	0	0
2	ACA$ACACG**G**	**G**	0	1	1	0
3	ACACGGACA**$**	**$**	0	1	1	1
4	ACGGACA$A**C**	**C**	1	1	1	1
5	CA$ACACGG**A**	**A**	**1**	**1**	**2**	**1**
6	CACGGACA$**A**	**A**	1	2	2	1
7	CGGACA$AC**A**	**A**	1	3	2	1
8	GACA$ACAC**G**	**G**	1	4	2	1
9	GGACA$ACA**C**	**C**	1	4	2	2
10	—	**—**	**1**	**4**	**3**	**2**
Offset (O)	—	**—**	**0**	**1**	**5**	**8**

Tools using BWTs in Exact Pattern Matching

- Alignment
  - Bowtie (2009) and BWA (2009)
  - Build a BWT of the reference genome (~2-3 GB)
  - Align:
    - Given a 100 base pair read
    - Cut into smaller seed pieces (i. e. four 25-mers)
    - Exact search for the pieces separately - very fast using BWT
    - Use local alignment (dynamic program) to extend initial seed alignments and account for errors
  - Bowtie2 (2011) and Tophat2 (2013) are still very prominent and fast aligners

BWTs and String Collections

- We have a BWT of single string with some functions
  - Recover suffix
  - O(k) search for an arbitrary k-mer
- What if you have multiple strings?
  - Concatenate strings together?
  - Build each one as a separate BWT and query each one?

Multi-string BWTs

- **MSBWT** - a BWT containing a string collection instead of just a single string
- Earliest: Mantaci *et al.* (2005), used concatenation approach
- Bauer *et al.* (2011) - proposed version we will discuss today

MSBWT Construction

- Naive Construction:
  - Create all rotations for all strings in the collection
  - Sort all rotations together (Suffix Array)
  - Store the last symbols in each suffix
- Strings are "cyclic"
  - Getting the predecessor always gets a suffix from the same string
  - Impossible to "jump" from one string to another

Index	Rotations	Sorted Suffixes	MSBWT
0	ACCA$	$ACC**A**	**A**
1	CCA$A	$CAA**A**	**A**
2	CA$AC	A$AC**C**	**C**
3	A$ACC	A$CA**A**	**A**
4	$ACCA	AA$C**A**	**A**
5	CAAA$	AAA$**C**	**C**
6	AAA$C	ACCA**$**	**$**
7	AA$CA	CA$A**C**	**C**
8	A$CAA	CAAA**$**	**$**
9	$CAAA	CCA$**A**	**A**

The multi-string BWT for strings "ACCA$" and "CAAA$".

MSBWT and FM-index

- Identical Definition
- Find *k*-mer “CA”
  - Initialize to full range (“”)
    low, high = 0, 10
  - Find occurrences of “A”
    low = O[‘A’]+F[low][‘A’] = 2+0 = 2
    high = O[‘A’]+F[high][‘A’] = 2+5 = 7
  - Find occurrences of “CA”
    low = O[‘C’]+F[low][‘C’] = 7+0 = 7
    high = O[‘C’]+F[high][‘C’] = 7+2 = 9

Index	Suffix Array	MSBWT	FM-index		
			$	A	C
0	$ACC**A**	**A**	0	0	0
1	$CAA**A**	**A**	0	1	0
2	A$AC**C**	**C**	0	2	0
3	A$CA**A**	**A**	0	2	1
4	AA$C**A**	**A**	0	3	1
5	AAA$**C**	**C**	0	4	1
6	ACCA**$**	**$**	0	4	2
7	CA$A**C**	**C**	1	4	2
8	CAAA**$**	**$**	1	4	3
9	CCA$**A**	**A**	2	4	3
10	—	**–**	**2**	**5**	**3**
Offset (O)	—	**–**	**0**	**2**	**7**

Compression of sequencing datasets

- Using Run-length encoding again
- Reasons we expect compression:
  - True genomic repeats: gene families, long repeats, etc.
  - Over-sampling: 30x coverage means we expect 30 copies of every *k*-mer pattern
  - Sequencing errors may break up runs
  - Technical errors may cause biases for or against a particular pattern
- Real Mouse DNA-seq:
  - ~350 Giga-bases
  - ~41 GB using RLE (0.94 bits/base)
- Real Mouse RNA-seq:
  - ~8.9 Giga-bases
  - ~1.2 GB using RLE (1.05 bits/base)

MSBWT Applications

- Instead of building a BWT of a reference genome, build a MSBWT of the sequenced reads
- Arbitrary exact match *k*-mer queries
  - O(*k*) time
  - Enables fast searches/counting
- Recover an arbitrary read of length *L* from MSBWT
  - O(*L*) time
  - Enables extraction of user-selected reads

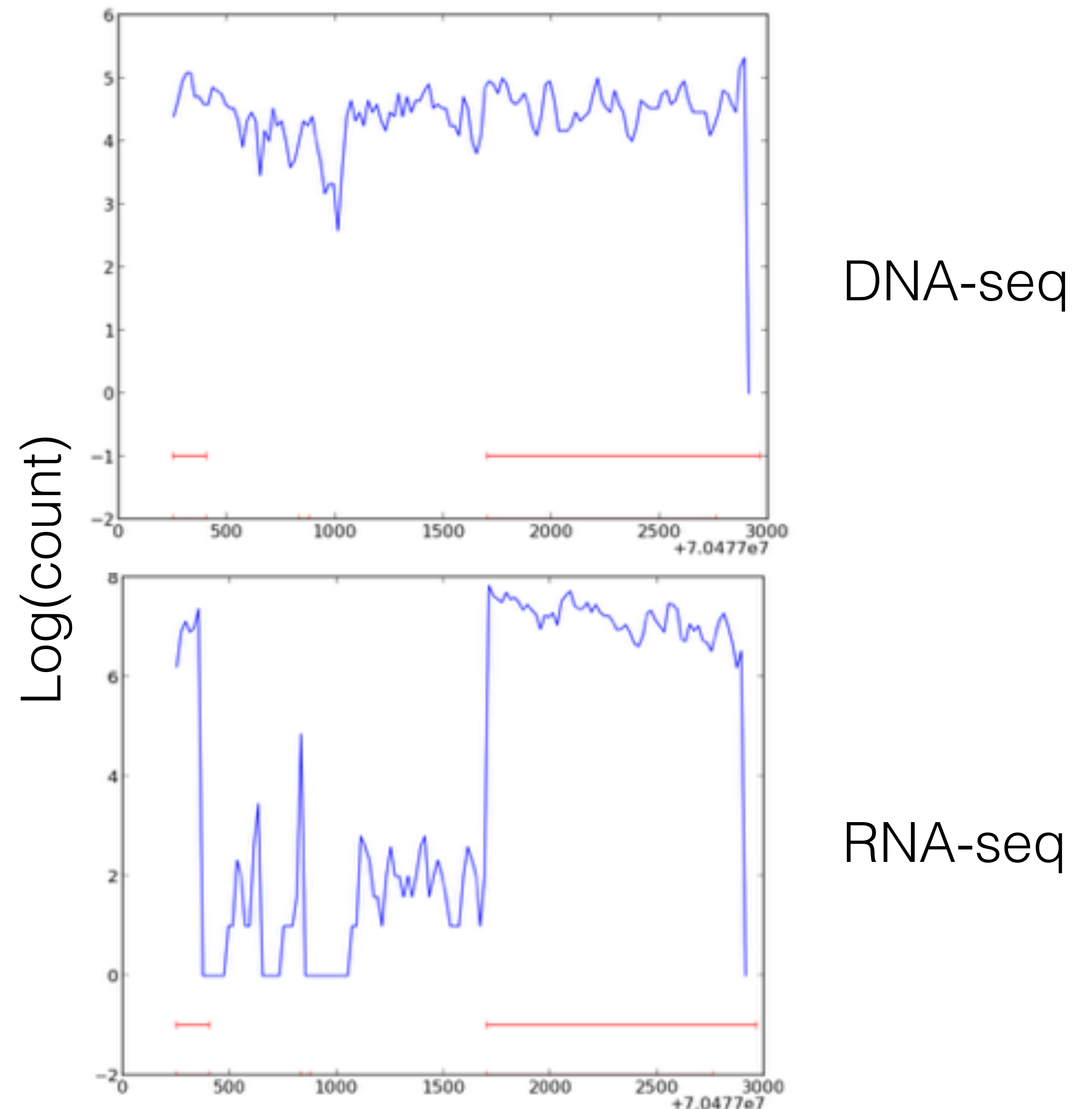
K-mer Search & Extraction

- Basic utilization
  - Search for all reads with a given *k*-mer
  - Extract all reads with that *k*-mer or the reverse-complement of the *k*-mer
  - Build a consensus

k-mer search & consensus builder
Green - *k*-mer query
Red - forward reads
Blue - reverse complemented reads

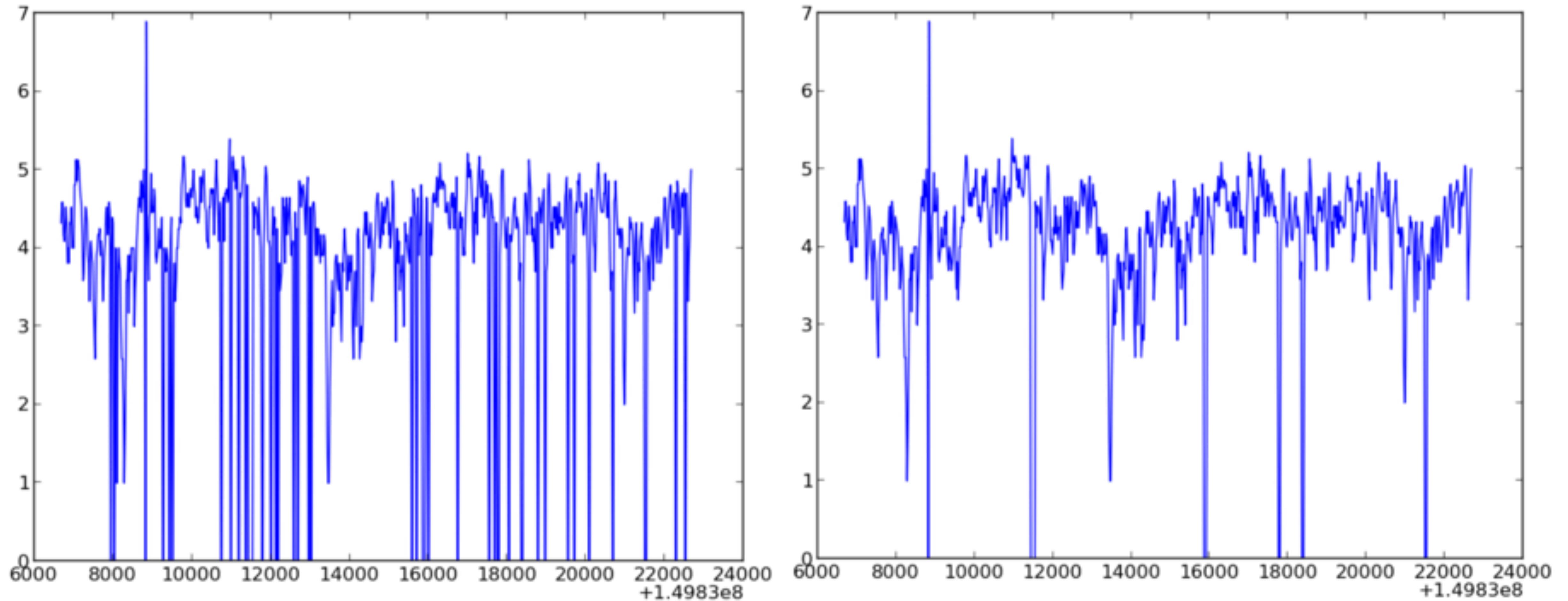
Reference-based Searches

- Given a reference genome and region of that genome
- Split reference into *k*-mers
- Count the abundance of each *k*-mer and plot
    - Fast - O(*k*) time per *k*-mer
    - Similar to a post-alignment pileup



CAST/EiJ at *Egr3*, counting 40-mers overlapping by 20

Reference Correction



Uncorrected   Corrected

```
149,838,013: 0   TTGATGGCTCGATGCATTCATTACCTGATCACTGCTCCCG
149,838,033: 0   TTACCTGATCACTGCTCCCGTTATGTAGGGAATGGGTACA

149,838,013: 18  TTGATGGCTCGATGCATTCATTACTTGATCACTGCTCCCG
149,838,033: 17  TTACTTGATCACTGCTCCCGTTATGTAGGGAATGGGTACA
```

CAST/EiJ DNA-seq for annotated gene *Igf2*

Targeted Assembly

- *De novo* assembly given a *k*-mer target known as the “seed” *k*-mer
- Extend the seed by counting the occurrence of each possible extension
- Generates a graph extending from the seed
  - Nodes - continuous unambiguous choice of extensions (similar to a contig)
  - Edges - multiple possible choices for extension

Targeted Assembly Tool Demo

- Demo used for CS events:
  - http://www.csbio.unc.edu/CEGSseq/index.py?run=msDemoTarget
- Gene
- Mitochondria

Summary

- Burrows-Wheeler Transform
  - Permutation of characters that represents a suffix array
  - Run-length encoded for compression
- FM-index
  - Derived from BWT
  - Exploits LF-mapping property
  - O(*k*) search time for arbitrary *k*-mer
  - Used in many fast aligners
- MSBWT
  - Applies to string collections
  - Enables database-like access to reads via *k*-mer searches