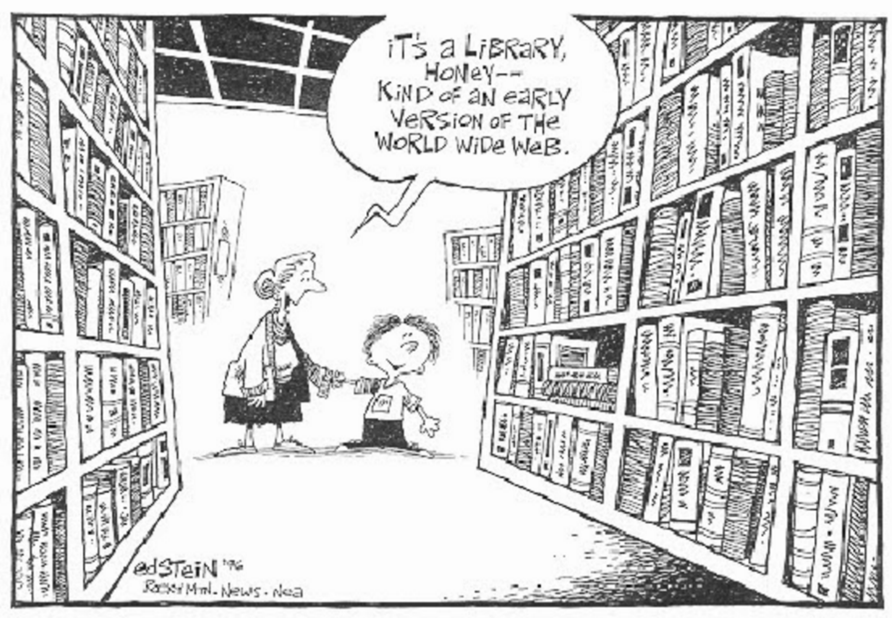
Multi-String BWTs



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

Analogy:

- Instead of searching for a substring within a single book, search every book of a library
  - Each book has it's own text, suffix array, and end-of-text delimiter
  - Searching allows us to find how many times a substring appears and in which texts

Bioinformatics?

- Search all genomes? You could, but that's not the main application.
- Search multiple chromosomes of an organism? You should, but even that is not the killer app

Naive Construction

- Create all rotations for all strings in the collection
- Sort all rotations together (Suffix Array)
- Store the predecessor of each suffix
- Strings are “cyclic”
- The predecessor is always from the same string
- Impossible to “jump” from one string to another

```
String1      Sorted     MSBWT
ACCA$        $ACCA        A
CCA$A        $CAAA        A
CA$AC        A$ACC        C
A$ACC        A$CAA        A
$ACCA        AA$CA        A
             AAA$C        C
String2      ACCA$        $
CAAA$        CA$AC        C
AAA$C        CAAA$        $
AA$CA        CCA$A        A
A$CAA
$CAAA
```

MSBWT's FM-index

Identical Definition

- Find k-mer “CA”
- Initialize to full range ('')
- lo, hi = 0, 10
- Find occurrences of 'A'
  - lo = Offset['A'] + FMindex[lo]['A'] = 2 + 0 = 2
  - hi = Offset['A'] + FMindex[hi]['A'] = 2 + 5 = 7
- Find occurrences of “CA”
  - lo = Offset['C'] + FMindex[lo]['C'] = 7 + 0 = 7
  - hi = Offset['C'] + FMindex[hi]['C'] = 7 + 2 = 9

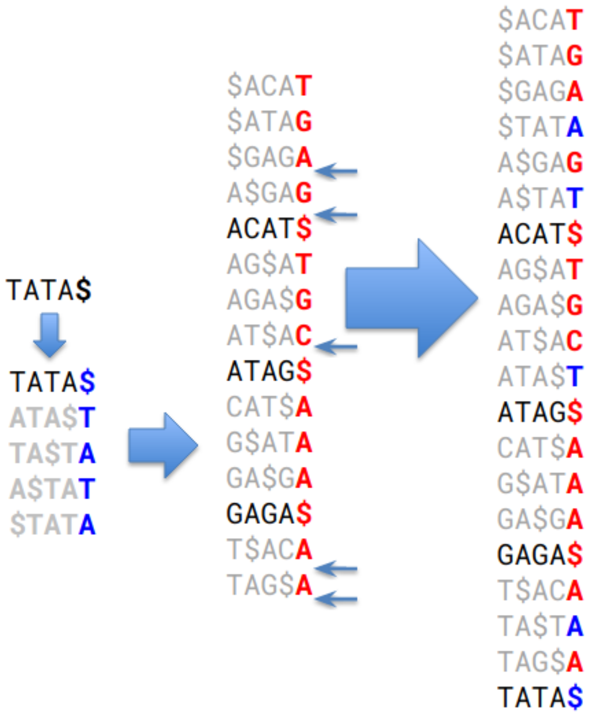
```
                                    FM-index
String1       Sorted     MSBWT        $  A  C
ACCA$         $ACCA        A      0:  0  0  0
CCA$A         $CAAA        A      1:  0  1  0
CA$AC         A$ACC        C      2:  0  2  0
A$ACC         A$CAA        A      3:  0  2  1
$ACCA         AA$CA        A      4:  0  3  1
              AAA$C        C      5:  0  4  1
String2       ACCA$        $      6:  0  4  2
CAAA$         CA$AC        C      7:  1  4  2
AAA$C         CAAA$        $      8:  1  4  3
AA$CA         CCA$A        A      9:  2  4  3
A$CAA                            10:  2  5  3
$CAAA                        Offset:  0  2  7
```

- Searching and extracting suffixes are identical to a BWT

Incremental MSBWT Construction

- A key tool missing from the BWTs toolbox-- *adding new strings to an existing msBWT*

- You could reconstruct the suffix array of the msBWT using suffix(i, fmindex) for all i, and then insert the suffixes of the new string.
- Variant of find(); Find the insertion point of new string's jth suffix, s_j
- Add last character to msBWT
- Update the FMindex



TATA$

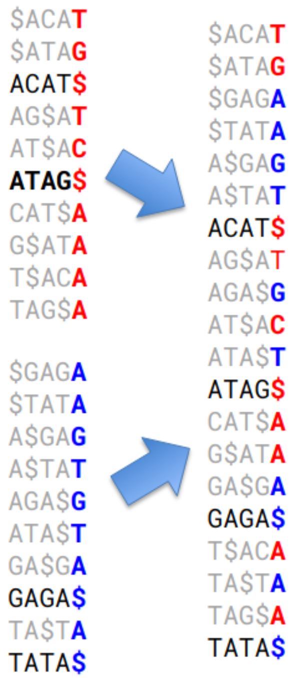
TATA$
ATA$T
TA$TA
A$TAT
$TATA

$ACAT
$ATAG
$GAGA
A$GAG
ACAT$
AG$AT
AGA$G
AT$AC
ATAG$
CAT$A
G$ATA
GA$GA
GAGA$
T$ACA
TAG$A

$ACAT
$ATAG
$GAGA
$TATA
A$GAG
A$TAT
ACAT$
AG$AT
AGA$G
AT$AC
ATA$T
ATAG$
CAT$A
G$ATA
GA$GA
GAGA$
T$ACA
TA$TA
TAG$A
TATA$

Merging msBWTs

- BETTER YET! Rather than inserting new strings, build a BWT of the new strings and merge the new and old BWTs
- Suffixes of BTWs are already sorted
- BTWs are interleaved
- In the worse case (ties) the entire suffix must be considered, but general the longest common prefix of suffixes is smaller
- Minimal overhead
- Well suited for divide an conquer approaches (like merge sort)
- Easy to merge multiple data sets!
- Compression improves!



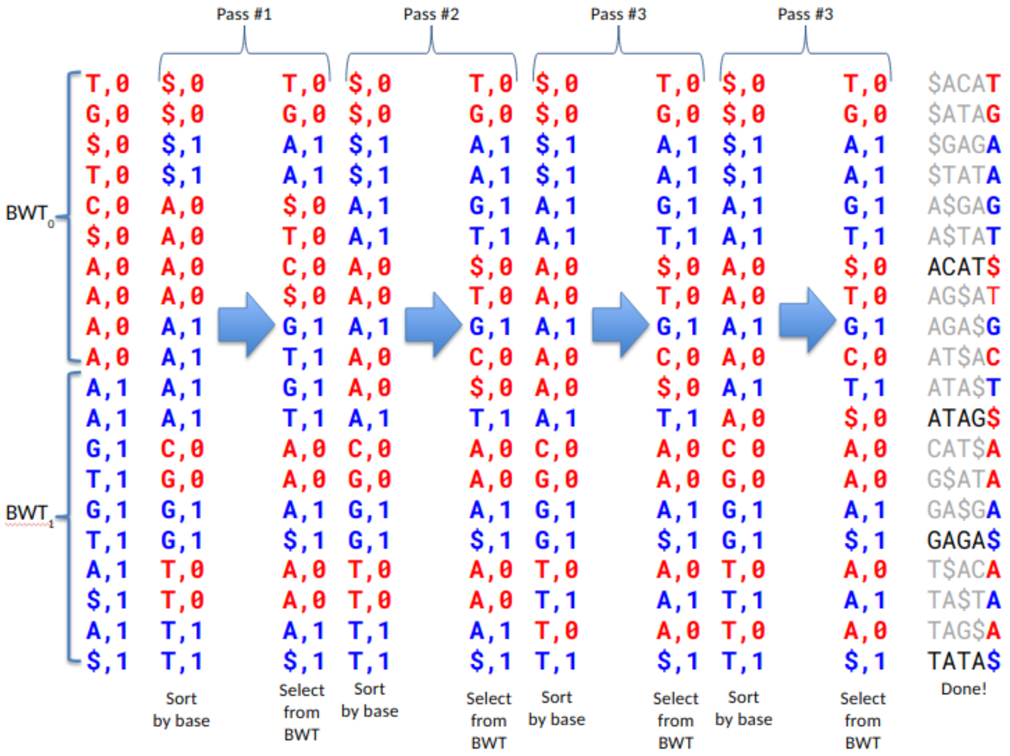
$ACAT
$ATAG
ACAT$
AG$AT
AT$AC
ATAG$
CAT$A
G$ATA
T$ACA
TAG$A

$GAGA
$TATA
A$GAG
A$TAT
AGA$G
ATA$T
GA$GA
GAGA$
TA$TA
TATA$

$ACAT
$ATAG
$GAGA
$TATA
A$GAG
A$TAT
ACAT$
AG$AT
AGA$G
AT$AC
ATA$T
ATAG$
CAT$A
G$ATA
GA$GA
GAGA$
T$ACA
TA$TA
TAG$A
TATA$

Merging Steps

- msBWT merging alternates between sorting and interleaving



		Pass #1		Pass #2		Pass #3		Pass #3		
BWT_0	T,0	$,0	T,0	$,0	T,0	$,0	T,0	$,0	T,0	$ACAT
	G,0	$,0	G,0	$,0	G,0	$,0	G,0	$,0	G,0	$ATAG
	$,0	$,1	A,1	$,1	A,1	$,1	A,1	$,1	A,1	$GAGA
	T,0	$,1	A,1	$,1	A,1	$,1	A,1	$,1	A,1	$TATA
	C,0	A,0	$,0	A,1	G,1	A,1	G,1	A,1	G,1	A$GAG
	$,0	A,0	T,0	A,1	T,1	A,1	T,1	A,1	T,1	A$TAT
	A,0	A,0	C,0	A,0	$,0	A,0	$,0	A,0	$,0	ACAT$
	A,0	A,0	$,0	A,0	T,0	A,0	T,0	A,0	T,0	AG$AT
	A,0	A,1	G,1	A,1	G,1	A,1	G,1	A,1	G,1	AGA$G
	A,0	A,1	T,1	A,0	C,0	A,0	C,0	A,0	C,0	AT$AC
BWT_1	A,1	A,1	G,1	A,0	$,0	A,0	$,0	A,1	T,1	ATA$T
	A,1	A,1	T,1	A,1	T,1	A,1	T,1	A,0	$,0	ATAG$
	G,1	C,0	A,0	C,0	A,0	C,0	A,0	C 0	A,0	CAT$A
	T,1	G,0	A,0	G,0	A,0	G,0	A,0	G,0	A,0	G$ATA
	G,1	G,1	A,1	G,1	A,1	G,1	A,1	G,1	A,1	GA$GA
	T,1	G,1	$,1	G,1	$,1	G,1	$,1	G,1	$,1	GAGA$
	A,1	T,0	A,0	T,0	A,0	T,0	A,0	T,0	A,0	T$ACA
	$,1	T,0	A,0	T,0	A,0	T,1	A,1	T,1	A,1	TA$TA
	A,1	T,1	A,1	T,1	A,1	T,0	A,0	T,0	A,0	TAG$A
	$,1	T,1	$,1	T,1	$,1	T,1	$,1	T,1	$,1	TATA$
		Sort by base	Select from BWT	Sort by base	Select from BWT	Sort by base	Select from BWT	Sort by base	Select from BWT	Done!

In Python

```python
def mergeBWT(bwt1, bwt2):
    interleave = [(c, 0) for c in bwt1] + [(c, 1) for c in bwt2]
    passes = min(len(bwt1), len(bwt2))
    for p in xrange(passes):
        i, j = 0, 0
        nextInterleave = []
        for c, k in sorted(interleave, key=lambda x: x[0]):
            if (k == 0):
                b = bwt1[i]
                i += 1
            else:
                b = bwt2[j]
                j += 1
            nextInterleave.append((b, k))
        if (nextInterleave == interleave):
            break
        interleave = nextInterleave
    return ''.join([c for c, k in interleave])

bwt1 = "TG$TC$AAAA"
bwt2 = "AAGTGTA$A$"
bwt12 = mergeBWT(bwt1, bwt2)
print bwt12
FM, Offset = FMIndex(bwt12)
for i in xrange(len(bwt12)):
    j = (i>>2)+(i&3)*(len(bwt12)/4)
    print "%2d: %s" % (j, recoverSuffix(j,bwt12,FM,Offset)), "\n" if (i % 4 == 3) else "",
```

```
TGAAGT$TGCT$AAA$AAA$
 0: $ACAT   5: A$TAT  10: ATA$T  15: GAGA$
 1: $ATAG   6: ACAT$  11: ATAG$  16: T$ACA
 2: $GAGA   7: AG$AT  12: CAT$A  17: TA$TA
 3: $TATA   8: AGA$G  13: G$ATA  18: TAG$A
 4: A$GAG   9: AT$AC  14: GA$GA  19: TATA$
```

MSBWT Applications

- Instead of building a BWT of a reference genome, build a MSBWT of every 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

Compression of high-throughput sequencing

- 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:
  - 368654191 × 151 × 2 = ~112 Giga-bases
  - Compresses to ~15.3 GB using RLE (1.09 bits/base)
- Real Mouse RNA-seq:
  - ~8.9 Giga-bases
  - ~1.2 GB using RLE (1.05 bits/base)

K-mer Search & Extraction

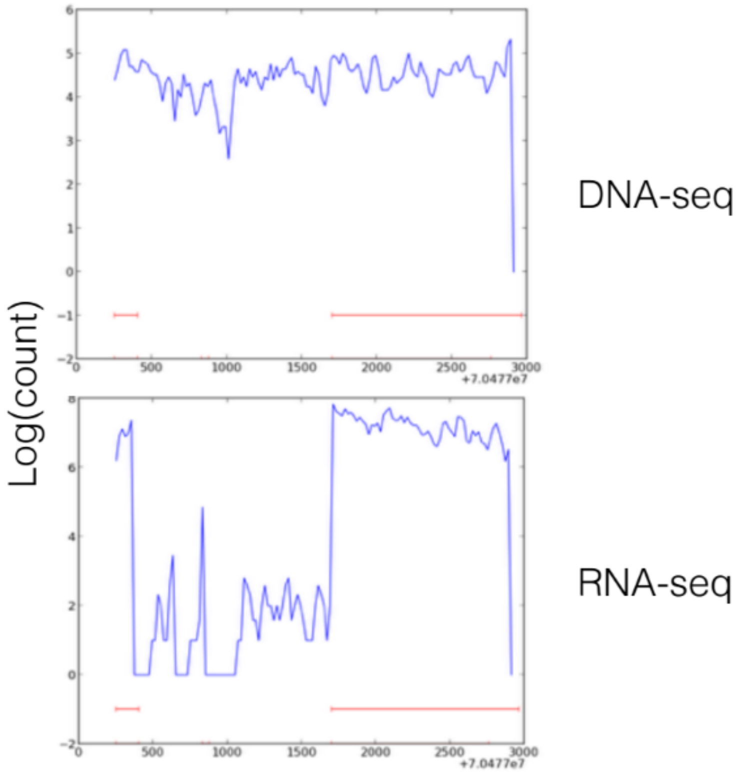
Basic Use:

Green: query k-mer. Red: forward reads. Blue: reverse-complement reads.

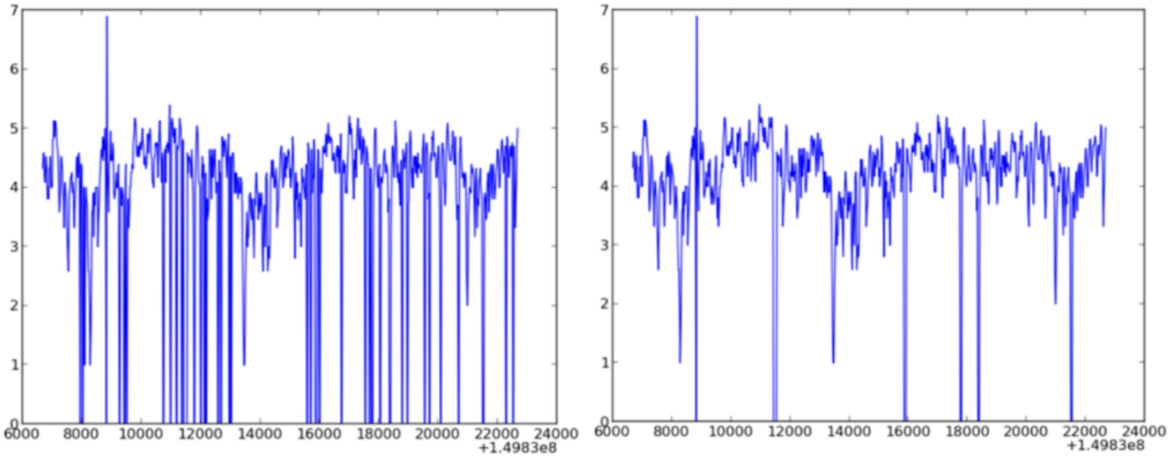
- Search for all reads with a given k-mer
- Extract all reads with that k-mer and its reverse-complement
- Build a consensus

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



Reference Correction



Uncorrected

Corrected

```
149,838,013: 0  TTGATGGCTCGATGCATTCATTACCTGATCACTGCTCCCG
149,838,033: 0  TTACCTGATCACTGCTCCCGTTATGTAGGGAATGGGTACA
                            ↓
149,838,013: 18 TTGATGGCTCGATGCATTCATTACTTGATCACTGCTCCCG
149,838,033: 17 TTACTTGATCACTGCTCCCGTTATGTAGGGAATGGGTACA
```

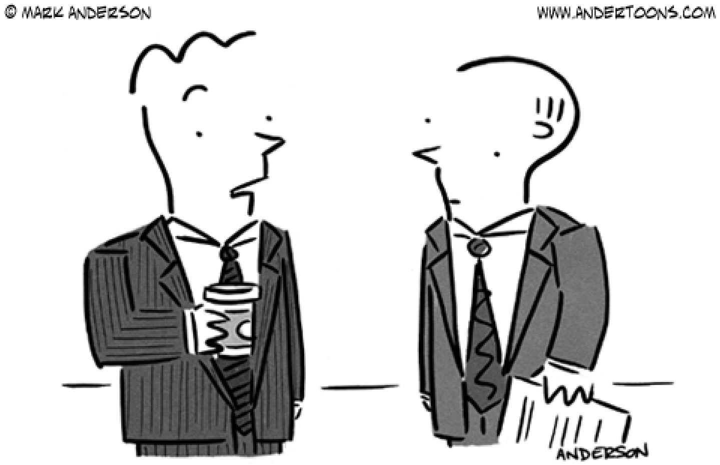
CAST/EiJ DNA-seq for annotated gene *Igf2*

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, independent of BWT's size
    - Used in many fast aligners
- MSBWT
    - Applies to string collections
    - Enables database-like access to reads via k-mer searches

Next Time

Sequence Alignment



"You ever have one of those days where you just don't feel like aligning?"