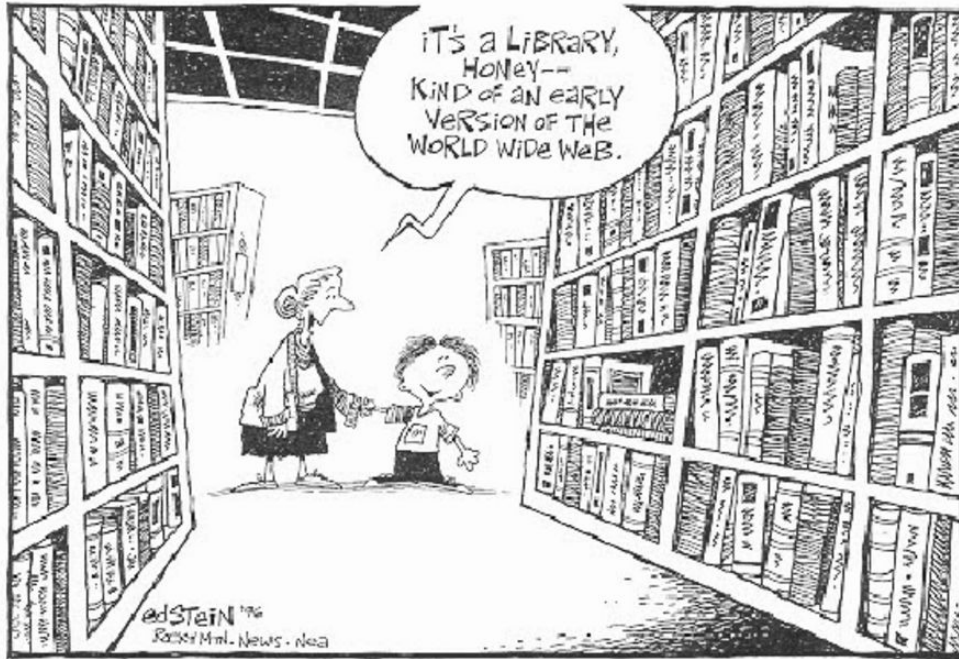
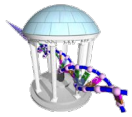


Comp 555 - BioAlgorithms - Spring 2021



- **WELLNESS DAY THIS THURSDAY**
- **PROBLEM SET #1 IS GRADED**
- **PROBLEM SET #3 IS DUE 3/16**
- **MIDTERM IS ON 3/18**

Multi-String BWTs

MSBWT



A BWT of a ***string collection*** instead of just a single string

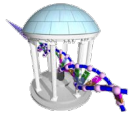
- Earliest: Mantaci et al. (2005), used concatenation approach
- Bauer et al. (2011) - proposed version we will discuss today

Analogy:

- Instead of searching for a substring within a single book, search every book of a library
 - Each book has its 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
- Strings can have different lengths

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		

(Unsorted suffixes) (Merged and sorted) (multi-string BWT
Note the 2 '\$s')

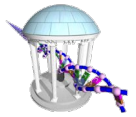
MSBWT's FM-index



Identical Definition

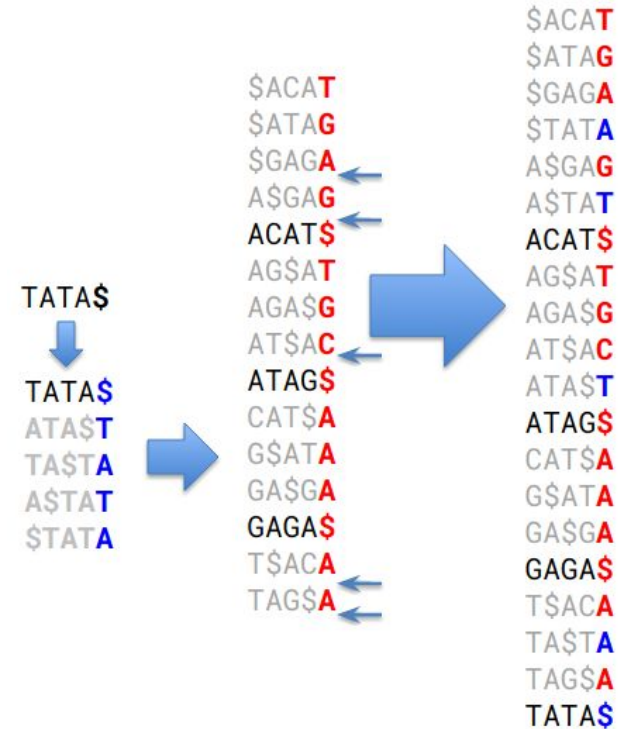
- Find k-mer "CA"
- Initialize to full range ("")
- lo, hi = 0, 10
- Find occurrences of 'A'
 - $lo = \text{Offset}['A'] + \text{FMindex}[lo]['A'] = 2 + 0 = 2$
 - $hi = \text{Offset}['A'] + \text{FMindex}[hi]['A'] = 2 + 5 = 7$
- Find occurrences of "CA"
 - $lo = \text{Offset}['C'] + \text{FMindex}[lo]['C'] = 7 + 0 = 7$
 - $hi = \text{Offset}['C'] + \text{FMindex}[hi]['C'] = 7 + 2 = 9$
- Searching and extracting suffixes are identical to a BWT

	String1	Sorted	MSBWT	FM-index
				\$ A C
	ACCA\$	\$ACCA	A	0: 0 0 0
	CCAS\$	\$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

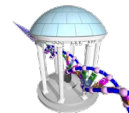


Incremental MSBWT Construction

- 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 $\text{suffix}(i, \text{findex})$ for all i , and then insert the suffixes of the new string.
- Variant of $\text{find}()$; Find the insertion point of new string's j^{th} suffix, s_j
- Add last character to msBWT
- Update the FMindex



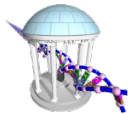
Our original BWT code



```
In [8]: def FMIndex(bwt):
    fm = [{c: 0 for c in bwt}]
    for c in bwt:
        row = {symbol: count + 1 if (symbol == c) else count for symbol, count in fm[-1].items()}
        fm.append(row)
    offset = {}
    N = 0
    for symbol in sorted(row.keys()):
        offset[symbol] = N
        N += row[symbol]
    return fm, offset

def recoverSuffix(i, BWT, FMIndex, Offset):
    suffix = ''
    c = BWT[i]
    predec = Offset[c] + FMIndex[i][c]
    suffix = c + suffix
    while (predec != i):
        c = BWT[predec]
        predec = Offset[c] + FMIndex[predec][c]
        suffix = c + suffix
    return suffix

def findBWT(pattern, FMIndex, Offset):
    lo = 0
    hi = len(FMIndex) - 1
    for symbol in reversed(pattern):
        lo = Offset[symbol] + FMIndex[lo][symbol]
        hi = Offset[symbol] + FMIndex[hi][symbol]
    return lo, hi
```



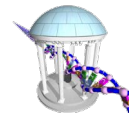
Inserting a new BWT into an existing msBWT

```
In [9]: 1 # Constructing a multistring BWT one suffix at a time
2 # first let's take a look at the implicit suffix array
3 bwt = "TGAG$TGC$AAA$AA"
4 fm, off = FMIndex(bwt)
5 for i in range(len(bwt)):
6     print("%2d: %s" % (i, recoverSuffix(i, bwt, fm, off)))
7 print()
8
9 # New string to include
10 new = "TATA$"
11 inserts = []
12 for i in range(len(new)):
13     rotation = new[i:]+new[:i]
14     l, h = findBWT(rotation, fm, off)
15     inserts.append((h, new[i-1], rotation))
16 print(inserts)
17
18 # Insert into original BWT in reverse order
19 for i, c, rot in sorted(inserts, reverse=True):
20     bwt = bwt[:i] + c + bwt[i:]
21
22 # Look at result
23 print(bwt)
24 print()
25 fm, off = FMIndex(bwt)
26 for i in range(len(bwt)):
27     print("%2d: %s" % (i, recoverSuffix(i, bwt, fm, off)))
```

```
0: $ACAT
1: $ATAG
2: $GAGA
3: $GAG
4: ACAT$
5: AG$AT
6: AG$G
7: AT$AC
8: ATAG$
9: CAT$A
10: G$ATA
11: GA$GA
12: GAG$A
13: T$ACA
14: TAG$A
```

1. How many strings are in this msBWT?
2. What strings are in this msBWT?
3. How could you have figured this out from the msBWT alone?

Before and After



```
[(15, '$', 'TATA$'), (8, 'T', 'ATA$T'), (14, 'A', 'TA$TA'), (4, 'T', 'A$TAT'), (3, 'A', '$TATA')]
TGAAGT$TGCT$AAA$AAA$
```

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

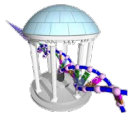
Problem: Add the string "TATA\$" to an existing msBWT.

Implies 5 new suffixes: TATA\$, ATA\$T, TA\$TA, A\$TAT, \$TATA

Insert these into existing msBWT. Use BWT search to find all, all insert positions first, then insert from highest to lowest index.

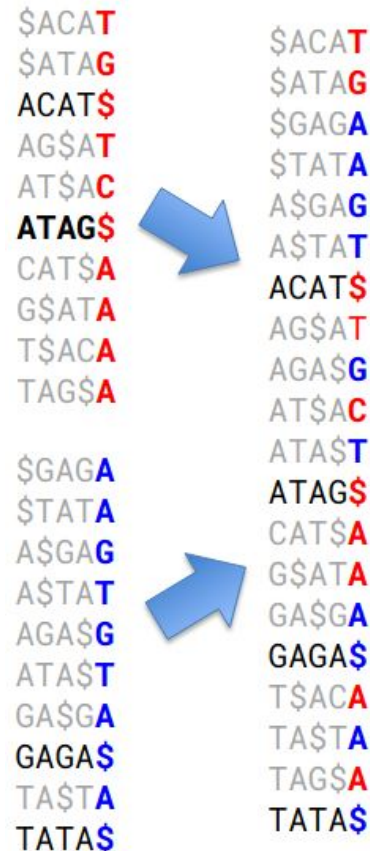
That seems a little tricky...

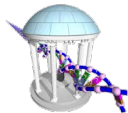




Merging msBWTs

- 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!

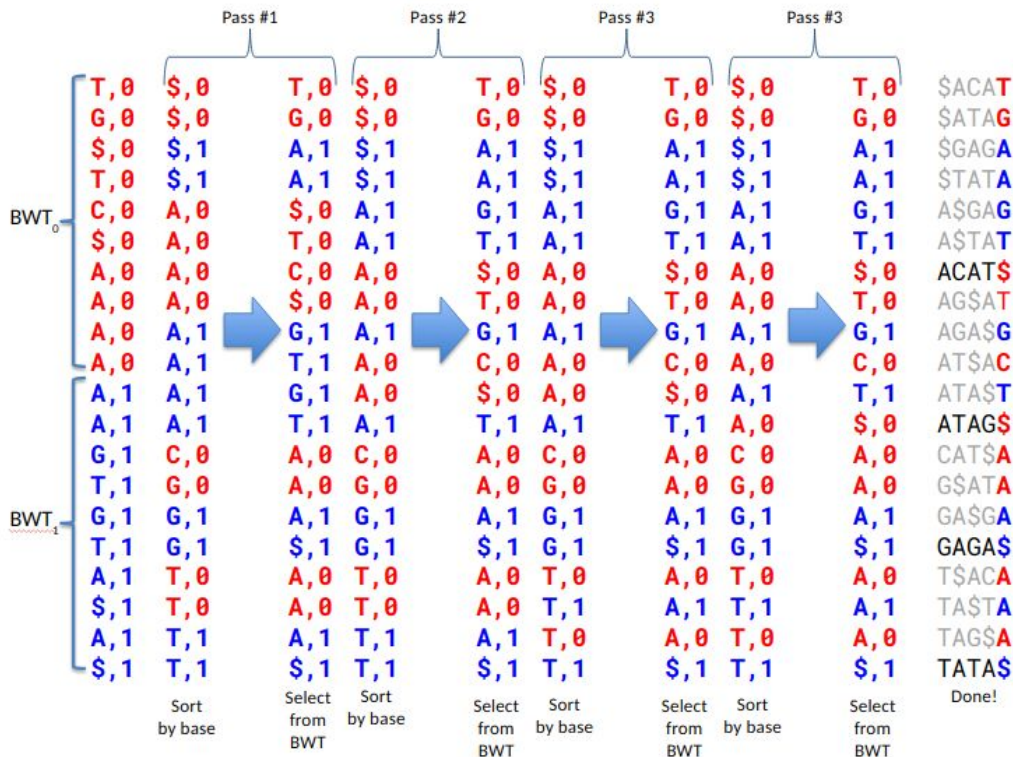




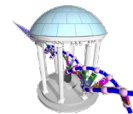
Merging Steps

msBWT merging alternates between sorting and interleaving

1. Consider the BWTs as a tuple of (character, BWTid) pairs
2. Sort these tuples
3. Based on the BWTids after the sort, select a new character
For each tuple from the original msBWTs
4. Repeat from Step 2 until the sort is stable
5. The resulting characters are the merged msBWT
6. Number of passes is proportional to largest LCP value.



In Python



```
In [12]: ▶ 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 range(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 = "AAGTGTAA$A$"
bwt12 = mergeBWT(bwt1, bwt2)
print(bwt12)
FM, Offset = FMIndex(bwt12)
for i in range(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 "", end='')
```

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

MSBWT Applications

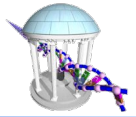


- Instead of building a BWT of a reference genome, build a MSBWT of every sequenced reads
- Arbitrary exact-match k-mer queries
- $O(k)$ time
- Enables fast searches/counting
- Recover an arbitrary read of length L from MSBWT
- $O(L)$ time
- Enables extraction of user-selected reads

Compression of high-throughput sequencing

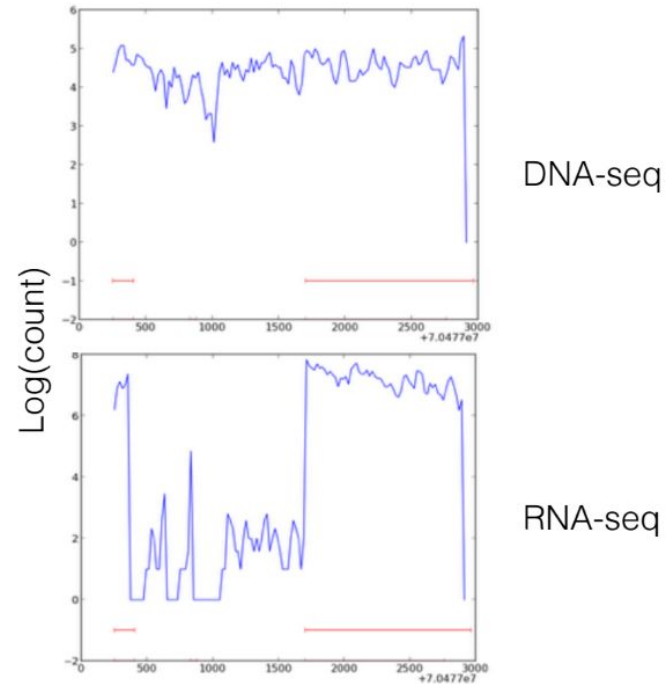


- Using Run-length encoding again
- Reasons we expect compression:
 - True genomic repeats: gene families, long repeats, etc.
 - Over-sampling: 30x coverage means we expect 30 copies of every k-mer pattern
- Sequencing errors may break up runs
- Technical errors may cause biases for or against a particular pattern
- Real Mouse DNA-seq:
 - $368654191 \times 151 \times 2 = \sim 112$ 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)

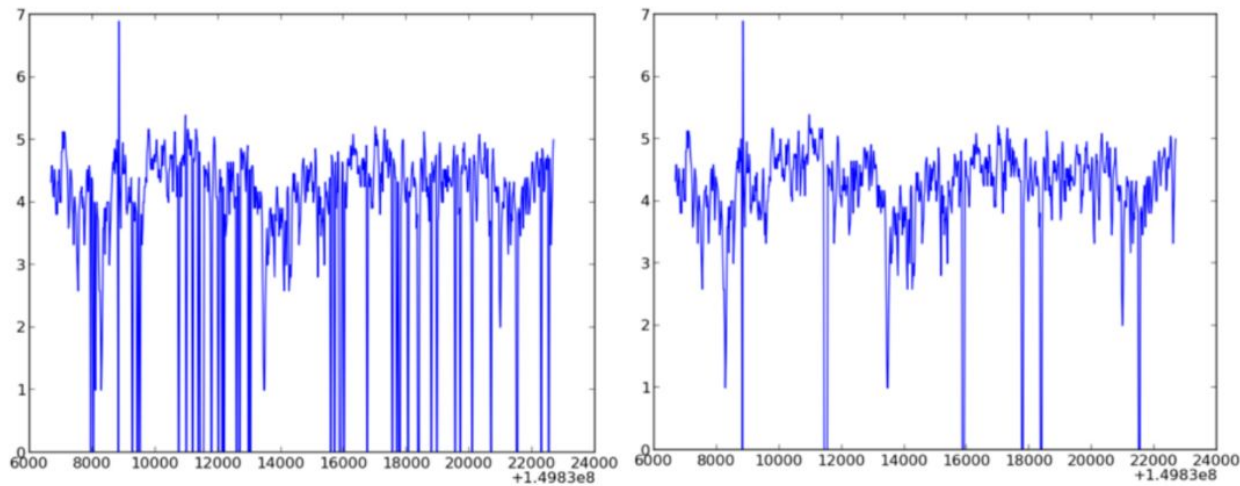
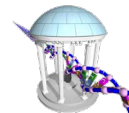


Reference-based Searches

- 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



Iterative Reference Correction



Uncorrected

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

Corrected

149,838,013: 18 TTGATGGCTCGATGCATTCATTACTTGATCACTGCTCCCG
149,838,033: 17 TTA~~CT~~TGATCACTGCTCCCGTTATGTAGGGAATGGGTACA

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