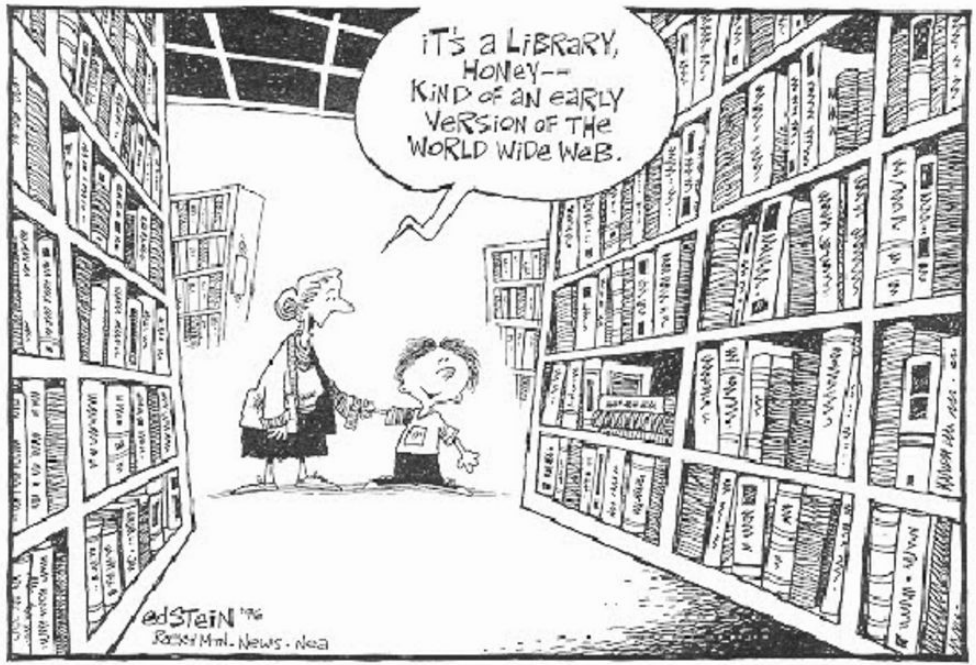
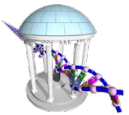
Comp 555 - BioAlgorithms - Spring 2018



- Problem set #2 is ONLINE
- Midterm is set for March 7

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

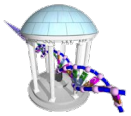
Analogy:

- 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

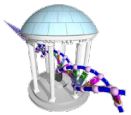
Naive Construction



- 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
- 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		

MSBWT's FM-index



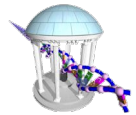
Identical Definition

- 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
- Searching and extracting suffixes are identical to a BWT

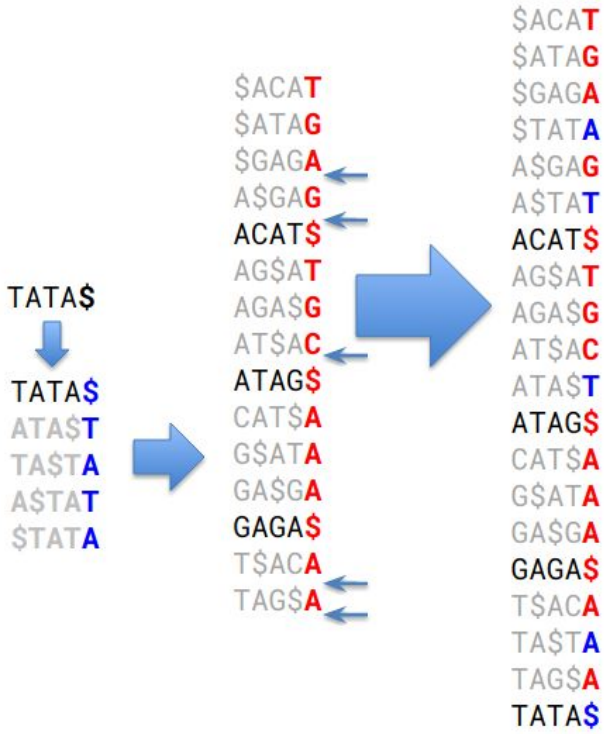
String1	Sorted	MSBWT
ACCA$	$ACCA	A
CCA$A	$CAAA	A
CA$AC	A$ACC	C
A$ACC	A$CAA	A
$ACCA	AA$CA	A
	AAA$C	C
String2	ACCA$	$
CAAA$	CA$AC	C
AAA$C	CAAA$	$
AA$CA	CCA$A	A
A$CAA		
$CAAA		

	FM-index		
	$	A	C
0:	0	0	0
1:	0	1	0
2:	0	2	0
3:	0	2	1
4:	0	3	1
5:	0	4	1
6:	0	4	2
7:	1	4	2
8:	1	4	3
9:	2	4	3
10:	2	5	3
Offset:	0	2	7

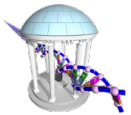
Incremental MSBWT Construction



- 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 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 j^{th} suffix, s_j
- Add last character to msBWT
- Update the FMindex



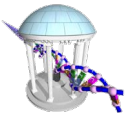
Our original BWT code



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



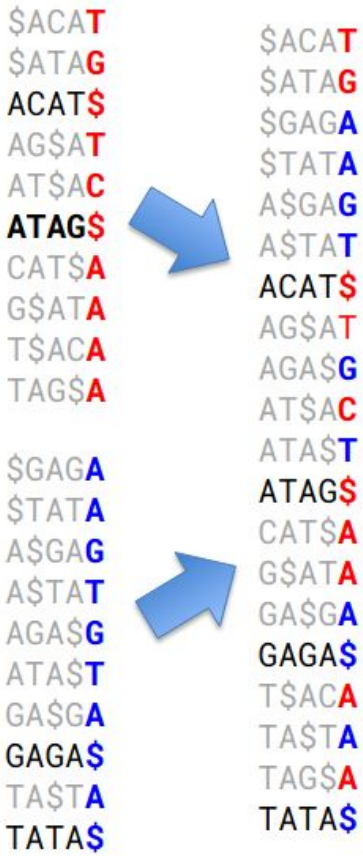
In [2]:

```
bwt = "TGAG$TGC$AAA$AA"
fm, off = FMIndex(bwt)
for i in range(len(bwt)):
    print("%2d: %s" % (i, recoverSuffix(i, bwt, fm, off)))

new = "TATA$"
inserts = []
for i in range(len(new)):
    l, h = findBWT(new[i:]+new[:i],fm,off)
    inserts.append((h,new[i-1]))

for i, c in sorted(inserts, reverse=True):
    bwt = bwt[:i] + c + bwt[i:]

print(bwt)
```

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

Merging msBWTs

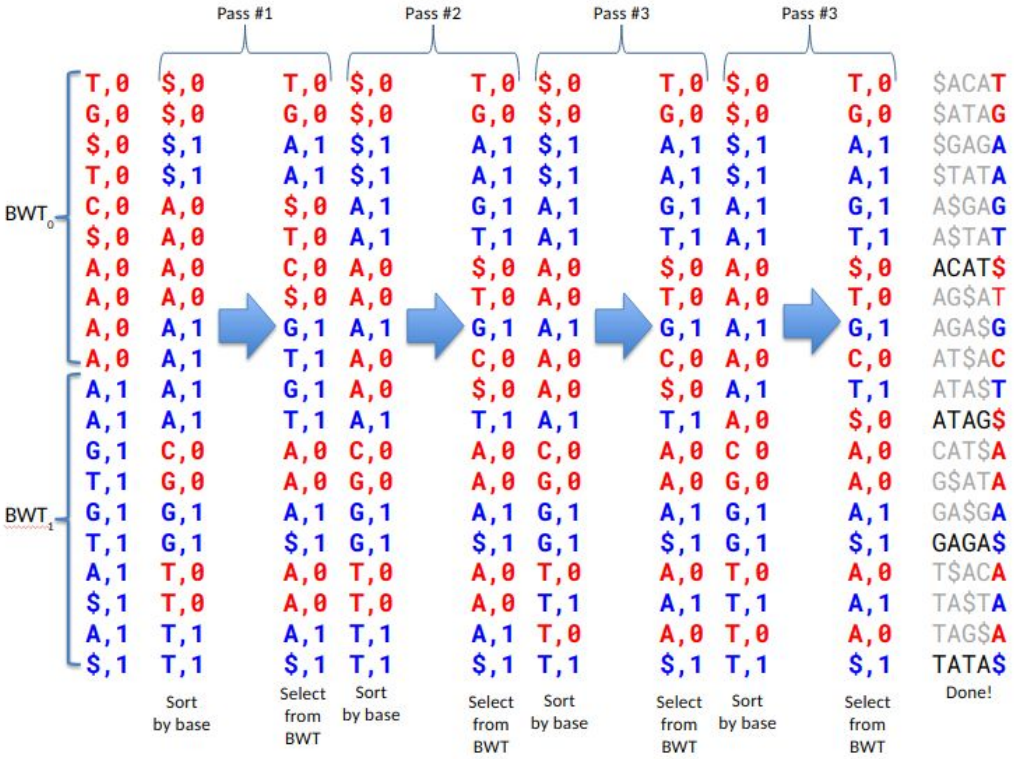
- BETTER YET! Rather than inserting new strings, build a BWT of the new strings and merge the new and old BWTs
- Suffixes of BTWs are already sorted
- BTWs are interleaved
- In the worse case (ties) the entire suffix must be considered, but general the longest common prefix of suffixes is smaller
- Minimal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



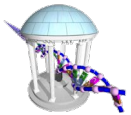
Merging Steps

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

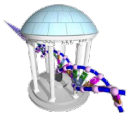


In [12]:

```
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 = "AAGTGTA$A$"
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='')
```

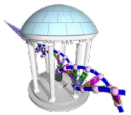
```
TGAAGT$TGCT$AAA$AAA$
 0: $ACAT  5: A$TAT 10: ATA$T 15: GAGA$
 1: $ATAG  6: ACAT$ 11: ATAG$ 16: T$ACA
 2: $GAGA  7: AG$AT 12: CAT$A 17: TA$TA
 3: $TATA  8: AGA$G 13: G$ATA 18: TAG$A
 4: A$GAG  9: AT$AC 14: GA$GA 19: TATA$
```

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

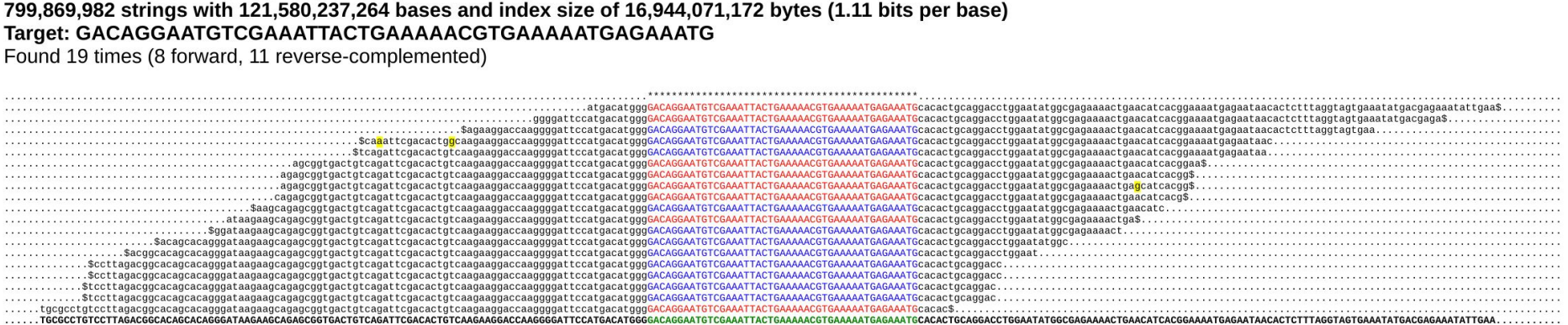


- Using Run-length encoding again
- Reasons we expect compression:
  - True genomic repeats: gene families, long repeats, etc.
  - Over-sampling: 30x coverage means we expect 30 copies of every k-mer pattern
- Sequencing errors may break up runs
- Technical errors may cause biases for or against a particular 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 & Read Extraction

Basic Use:

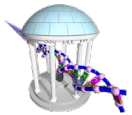
CC Genome: CC001M4363_UNC_NYGC
799,869,982 strings with 121,580,237,264 bases and index size of 16,944,071,172 bytes (1.11 bits per base)
Target: GACAGGAATGTCGAAATTACTGAAAAACGTGAAAAATGAGAAATG
Found 19 times (8 forward, 11 reverse-complemented)



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

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

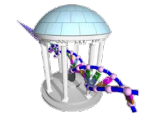
Moving over a little bit



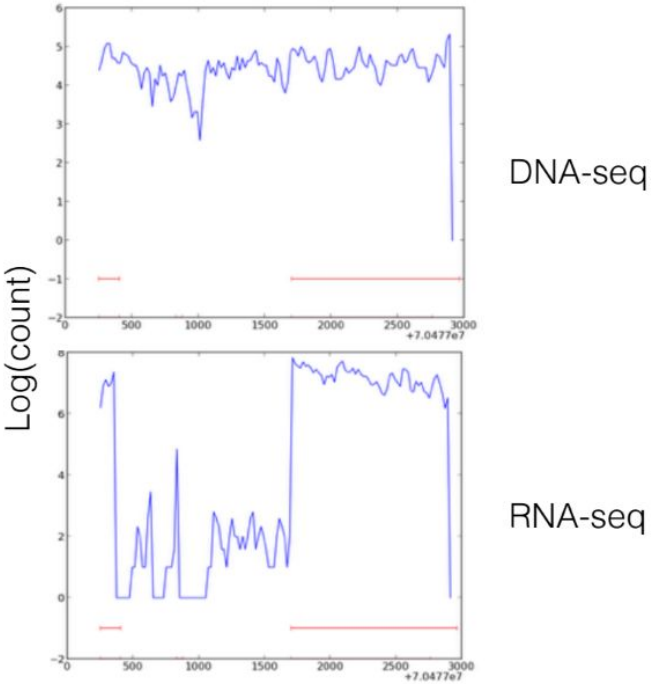
CC Genome: CC001M4363_UNC_NYGC
799,869,982 strings with 121,580,237,264 bases and index size of 16,944,071,172 bytes (1.11 bits per base)
Target: CGAAATTACTGAAAAACGTGAAAAATGAGAAATGCACACTGCAGGACCTGG
Found 46 times (25 forward, 21 reverse-complemented)

Moving over 12 bases shows a different story

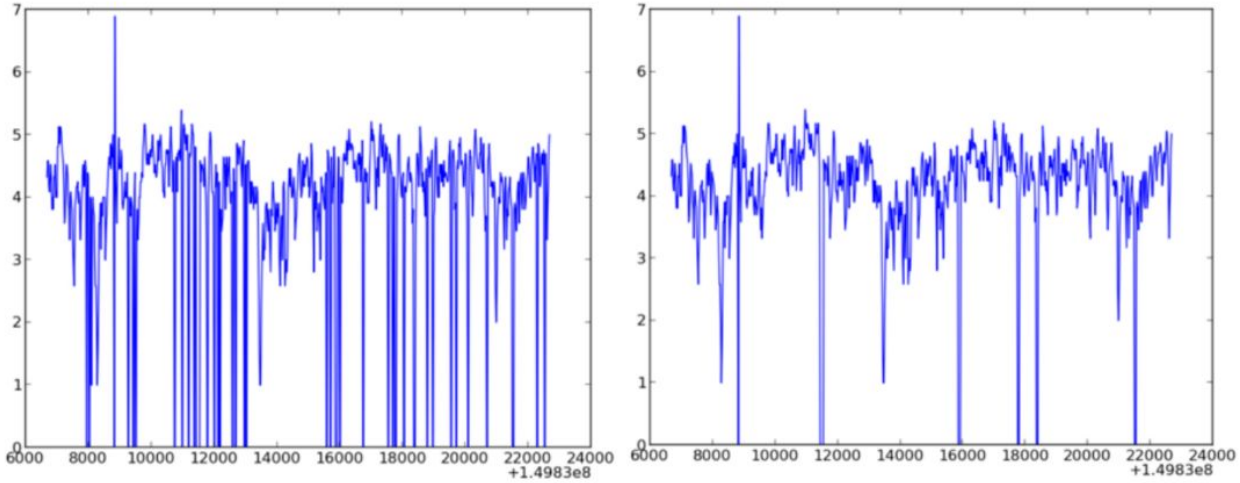
Reference-based Searches



- 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

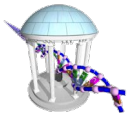


Iterative Reference Correction



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