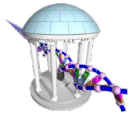


# Comp 555 - BioAlgorithms - Spring 2020



- **PROBLEM SET #2 IS DUE ON TUESDAY AND SOME PROBLEMS TAKE SIGNIFICANT TIME TO EXECUTE!**

Suffix Arrays and BWTs

# A tweak to argsort()

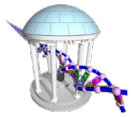


- Recall argsort() from last time:

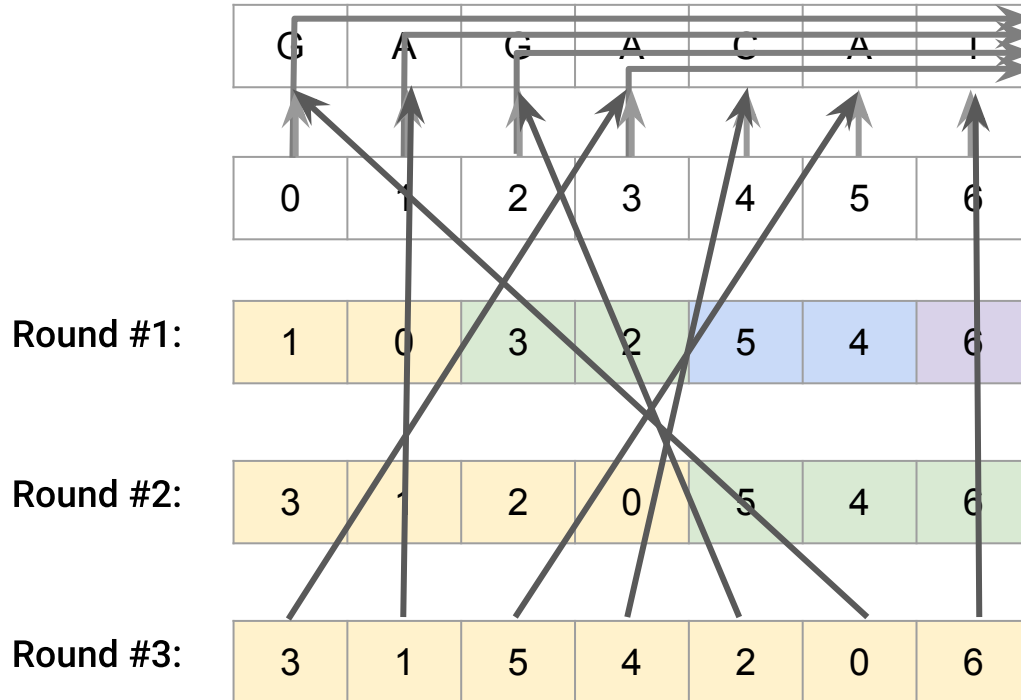
```
def argsort(input):  
    return sorted(range(len(input)), key=input.__getitem__)
```

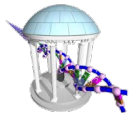
```
B = ["TAGACAT", "AGACAT", "GACAT", "ACAT", "CAT", "AT", "T"]  
print(argsort(B))
```

- If we know that our input is suffixes from a single string
  - the  $i^{\text{th}}$  suffix starts at index  $i$
  - thus we don't need to extract the suffixes, just use offsets



# Comparing suffixes in place





# Constructing a Suffix Array

```
In [20]: 1 def suffixArray(string):
2         return [i for i in sorted(range(len(string)), key=lambda x: string[x:])]
3
4 t = "amanaplanacanalpanama"
5 sa = suffixArray(t)
6 print(sa)
7 for i in sa:
8     print("%2d: %s" % (i, t[i:]))
```

```
[20, 9, 13, 18, 0, 7, 11, 16, 2, 4, 10, 6, 14, 19, 1, 8, 12, 17, 3, 15, 5]
20: a
9: acanalpanama
13: alpanama
18: ama
0: amanaplanacanalpanama
7: anacanalpanama
11: analpanama
16: anama
2: anaplanacanalpanama
4: aplanacanalpanama
10: canalpanama
6: lanacanalpanama
14: lpanama
19: ma
1: manaplanacanalpanama
8: nacanalpanama
12: nalpanama
17: nama
3: naplanacanalpanama
15: panama
5: planacanalpanama
```

The "key" parameter defines a "lightweight" function called a *lambda* function, which when given a value from the given list, the range in this instance, determines what should be used in the comparison of the sort, a substring from "string" in this instance.

# Searching a Suffix Array

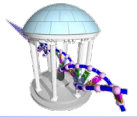


- Searching a sorted list requires  $O(\log(m))$  comparisons using **binary search**
- Each comparison is over  $n$  symbols of the pattern
- Thus, searching is  $O(n\log(m))$

```
In [6]: ▶ def findFirst(pattern, text, suffixarray):
        lo, hi = 0, len(text)
        while (lo < hi):
            middle = (lo+hi)//2
            if text[suffixarray[middle]:] < pattern:
                lo = middle + 1
            else:
                hi = middle
        return lo

        first = findFirst("an", t, sa)
        print(t)
        print(first, sa[first], t[sa[first]:])
```

```
amanaplanacanalpanama
5 7 anacanalpanama
```



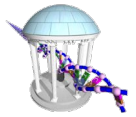
# Finding all Occurrences

A variant to binary search which finds the last occurrence of a pattern rather than the first. Only difference, uses "`<=`" instead of "`<`", but needs to trim string comparison to test for equality.

```
In [6]: def findLast(pattern, text, suffixarray):
        lo, hi = 0, len(text)
        while (lo < hi):
            middle = (lo+hi)//2
            if text[suffixarray[middle]:suffixarray[middle]+len(pattern)] <= pattern:
                lo = middle + 1
            else:
                hi = middle
        return lo

        print(t)
        last = findLast("an", t, sa)
        print(first, last)
        for suffix in sa[first:last]:          # recall "first" was found on the previous slide
            print("%3d: %s" % (suffix, t[suffix:]))
        print(last - first, "times")
```

```
amanaplanacanalpanama
5 9
  7: anacanalpanama
 11: analpanama
 16: anama
  2: anaplanacanalpanama
4 times
```



# Longest repeated substring?

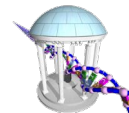
- Given a suffix array, we can compute a *helper* function, call the **Longest Common Prefix**, LCP

```
In [5]: 1 def computeLCP(text, suffixarray):
2         m = len(text)
3         lcp = [0 for i in range(m)]
4         for i in range(1,m):
5             u = suffixarray[i-1]
6             v = suffixarray[i]
7             n = 0
8             while text[u] == text[v]:
9                 n += 1
10                u += 1
11                v += 1
12                if (u >= m) or (v >= m):
13                    break
14                lcp[i] = n
15            return lcp
16
17 lcp = computeLCP(t, sa)
18
19 print("SA, LCP, Suffix")
20 for i, j in enumerate(sa):
21     print("%2d: %2d %s" % (j, lcp[i], t[j:]))
```

```
SA, LCP, Suffix
20: 0 a
9: 1 acanalpanama
13: 1 alpanama
18: 1 ama
0: 3 amanaplanacanalpanama
7: 1 anacanalpanama
11: 3 analpanama
16: 3 anama
2: 3 anaplanacanalpanama
4: 1 aplanacanalpanama
10: 0 canalpanama
6: 0 lanacanalpanama
14: 1 lpanama
19: 0 ma
1: 2 manaplanacanalpanama
8: 0 nacanalpanama
12: 2 nalpanama
17: 2 nama
3: 2 naplanacanalpanama
15: 0 panama
5: 1 planacanalpanama
```

- What is the longest repeated k-mer?
- How many distinct letters in alphabet?

# Summary to this point



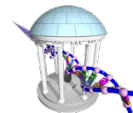
- Where:
  - $m$  is the length of the text to be searched
  - $n$  is the length of the pattern (maximum length if more than 1)
  - $p$  is the number of patterns

Method	Storage	Single Search	Multi Search
Brute Force	$O(m)$	$O(nm)$	$O(p n m)$
Keyword Tries	$O(pn)$	$O(nm)$	$O(p m)$
Suffix Trees	$O(m)^*$	$O(n)$	$O(p n)$
Suffix Arrays	$O(m)$	$O(n \log(m))$	$O(p n \log(m))$

\* With large constants



# A rather unknown compression approach



In 1994, two researchers from DEC research labs in Palo Alto, Michael Burrows and David Wheeler, devised a transformation for text that made it more compressible. Essentially, they devised a *invertible permutation* of any text that compresses well if it exhibits redundancy.

## Example:

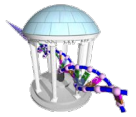
```
text = "amanaplanacanalpanama$"
BWT(text) = "amnnn$lcpmnapaaaaaala"
```

- Notice how the transformed text has long *runs* of repeated characters
- A simple form of compression, called run-length encoding, replaces repeated symbols by a (count, symbol) tuple
- If the count is 1, then just the symbol appears

Thus, the BWT(text) can be represented as:

```
Compress(BWT(text)) = am3n$lcpmnap7ala (16 chars instead of 22)
```

- The savings are even more impressive for longer strings
- Notice, they introduced a special "*end-of-text*" symbol (\$ in our case), which is lexicographically before any other symbol



# Key Idea behind the BWT

- Sorting Cyclical Suffixes (say that 3-times fast)

"Cyclical Suffixes"

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

"Sorted Cyclical Suffixes"

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

- The BWT of "tarheels" is the last column of the sorted cyclical suffixes "ltherea\$"
- Notice that the sorted cyclical suffixes have a lot in common with a suffix array.
- The BWT is just the "predecessor symbol of these suffixes", where "\$" precedes the first symbol

In [7]:

```
1 # making cyclical suffixes
2 t="carolina$"
3 print([t[i:]+t[:i] for i in range(len(t))])
```

```
['carolina$', 'arolina$c', 'rolina$ca', 'olina$car', 'lina$caro', 'ina$scarol', 'na$scaroli', 'a$scarolin', '$carolina']
```

# BWT in Python

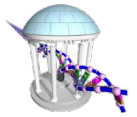


- Straightforward implementation based on the definition (there are faster construction methods)

```
In [28]: def BWT(t):
# create a sorted list of all cyclic suffixes of t
rotation = sorted([t[i:]+t[:i] for i in range(len(t))])
# concatenate the last symbols from each suffix
return ''.join(r[-1] for r in rotation)

print(BWT("banana$"))
print(BWT("amanaplanacanalpanama$"))
print(BWT("abanababab$"))
```

```
annb$a
amnnn$lcpmnapaaaaaala
nn$bnbaaaaa
```

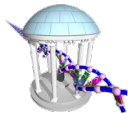


# BWT from a Suffix Array

- It is even simpler to compute the BWT from a Suffix Array
- Finds each suffix's "predecessor" symbol

```
In [9]: 1 def BWTfromSuffixArray(text, suffixarray):
        2     return ''.join(text[i-1] for i in suffixarray)
        3
        4 newt = 'carolina$'
        5 sa = suffixArray(newt)
        6 print(newt)
        7 print(sa)
        8 print(BWTfromSuffixArray(newt, sa))
```

```
carolina$
[8, 7, 1, 0, 5, 4, 6, 3, 2]
anc$loira
```



# Inverting a BWT

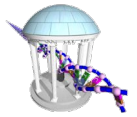
- A property of a transform is that there is no information loss-- they are invertible.

Algorithm: inverseBWT(bwt):

1. Create a table of  $\text{len}(\text{bwt})$  empty strings
2. repeat  $\text{length}(\text{bwt})$  times:
3.     prepend  $\text{bwt}$  as the first column of the table
4.     sort rows of the table alphabetically
5. return (row of table with bwt's 'EOF' character)

<u>0</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>
l	l\$	l\$t	l\$ta	l\$tar	l\$tarh	l\$tarhe	l\$tarhee	\$tarheel
t	ta	tar	tarh	tarhe	tarhee	tarheel	tarheel\$	arheel\$t
h	he	hee	heel	heel\$	heel\$t	heel\$ta	heel\$tar	eel\$tarh
e	ee	eel	eel\$	eel\$t	eel\$ta	eel\$tar	eel\$tarh	el\$tarhe
r	rh	rhe	rhee	rheel	rheel\$	rheel\$t	rheel\$ta	heel\$tar
e	el	el\$	el\$t	el\$ta	el\$tar	el\$tarh	el\$tarhe	l\$tarhee
a	ar	arh	arhe	arhee	arheel	arheel\$	arheel\$t	rheel\$ta
\$	\$t	\$ta	\$tar	\$tarh	\$tarhe	\$tarhee	\$tarheel	tarheel\$

- What else do you notice about the final table?



# Inverse BWT in Python

- Again, straightforward, but not the most efficient

```
In [33]: def inverseBWT(bwt):  
    # initialize the table from t  
    table = ['' for c in bwt]  
    for j in range(len(bwt)):  
        #insert the BWT as the first column  
        table = sorted([c+table[i] for i, c in enumerate(bwt)])  
    #return the row that ends with '$'  
    return table[bwt.index('$')]  
  
print(inverseBWT("ltherea$"))  
print(inverseBWT("amnnn$lcpmnapaaaaaala"))  
print(inverseBWT("annb$aa"))  
print(inverseBWT("nn$bnbaaaaa"))
```

```
tarheel$  
amanaplanacanalpanama$  
banana$  
abananaban$
```

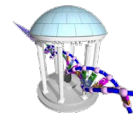
# BWT Compression



- Uncompressed the BWT(text) is same length as original text
- But, it has a tendency to form long runs of repeated symbols
- Why does it form runs?
- All suffixes of repeated substrings sort together and share predecessors
- Somewhere further down the BWT there is a series of suffixes starting with *u*'s that have *o*'s as predecessors
- Redundancy leads to compression

Suffixes from some text	BWT
⋮	⋮
ld you eat them in a box? ...	u
ld you eat them with a fox? ...	u
ld you like them here or there? ...	u
ld you like them in a house? ...	u
ld you like them with a mouse? ...	u
⋮	⋮
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
⋮	⋮

# What do BWTs have to do with searching strings?



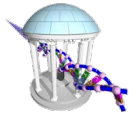
- There is close relationship between BWTs and Suffix Arrays
- We can construct a suffix array from a BWT as we saw with InverseBWT(bwt)
- Is there a way to access this “hidden” **implicit suffix array** for pattern searching?
- In 2005 two researchers, Ferragina & Manzini, figured out how
- First, an important property they uncovered

Snapshots at [jasonlove.com](http://jasonlove.com)



"Sir, we're just not reaching them. Only a small percentage of people own vinyl records, and hardly anyone thinks to play them backwards."



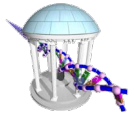


# Last-First (LF) mapping property

- The BWT transforms "banana\$" to "annb\$aa"
- The predecessor symbols of a suffix array preserve the relative suffix order
- The *j*th occurrence of a symbol in the BWT corresponds to its *j*th occurrence in the suffix array

\$banana	1st 'a' in BWT, 3rd 'a' in banana, 1st 'a' in suffix array
a\$banan	1st 'n' in BWT, 2nd 'n' in banana, 1st 'n' in suffix array
ana\$ban	2nd 'n' in BWT, 1st 'n' in banana, 2nd 'n' in suffix array
anana\$b	1st 'b' in BWT, 1st 'b' in banana, 1st 'b' in suffix array
banana\$	1st '\$' in BWT, 1st '\$' in banana, 1st '\$' in suffix array
na\$bana	2nd 'a' in BWT, 2nd 'a' in banana, 2nd 'a' in suffix array
nana\$b	3rd 'a' in BWT, 1st 'a' in banana, 3rd 'a' in suffix array

- This property allows one to traverse the suffix array indirectly
  - ex: The 1st "a" of the bwt is also the first "a" of the suffix array, and its predecessor is the 1st "n", whose predecessor is the 2nd "a", whose predecessor is the 2nd "n", and so on
- Meanwhile, the number of character occurrences in the BWT matches the suffix array (recall it is a permutation)



# The FM-index

- The FM-index is another *helper* data structure like the *LCP array* mentioned previously
- It is a 2D array whose size is  $[|text|+1, |\Sigma|]$ , where  $|\Sigma|$  is the alphabet size
- It keeps track of how many of each symbol have been seen in the BWT prior to its  $i$ th symbol
- The last  $m$  row is the totals for each symbol. By accumulating these totals you can determine the BWT index corresponding to the first of each symbol in the suffix array (Offset).
- Can be generated by a single scan through the BWT
- Memory overhead  $O(m|\Sigma|)$

Index	Suffix Array	BWT	FM-index			
			\$	a	b	n
0	\$banana	a	0	0	0	0
1	a\$banan	n	0	1	0	0
2	ana\$ban	n	0	1	0	1
3	anana\$b	b	0	1	0	2
4	banana\$	\$	0	1	1	2
5	na\$bana	a	1	1	1	2
6	nana\$ba	a	1	2	1	2
7	<b>Counts</b>		1	3	1	2
	<b>Offset</b>		0	1	4	5

# Constructing the FM-index

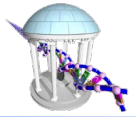


```
In [38]: def FMIndex(bwt):
          fm = [{c: 0 for c in bwt}] # a list of dictionaries
          for c in bwt:
              row = {symbol: count + 1 if (symbol == c) else count for symbol, count in fm[-1].items()}
              fm.append(row)
          offset = {}
          N = 0
          for symbol in sorted(row.keys()):
              offset[symbol] = N
              N += row[symbol]
          return fm, offset

          bwt = "annb$aa"
          FM, Offset = FMIndex(bwt)
          print("BWT %2s,%2s,%2s,%2s" % tuple([symbol for symbol in sorted(Offset.keys())]))
          for i, row in enumerate(FM):
              data = [bwt[i]+':' if i < len(bwt) else '']+ [row[symbol] for symbol in sorted(row.keys())]
              print("%3s %2d,%2d,%2d,%2d" % tuple(data))
          print()
          print([(sym, Offset[sym]) for sym in sorted(Offset.keys())])
```

```
BWT $, a, b, n
a: 0, 0, 0, 0
n: 0, 1, 0, 0
n: 0, 1, 0, 1
b: 0, 1, 0, 2
$: 0, 1, 1, 2
a: 1, 1, 1, 2
a: 1, 2, 1, 2
   1, 3, 1, 2
```

```
[('$', 0), ('a', 1), ('b', 4), ('n', 5)]
```



# Find a Suffix's Predecessor

- Given an index  $i$  in the BWT, find the index in the BWT of the suffix preceding the suffix represented by  $i$
- Suffix 5 is preceded by suffix 2
- Suffix 2 is preceded by suffix 6
- Suffix 6 is preceded by suffix 3
- The predecessor suffix of index  $i$ :

```
c = BWT[i]
predec = Offset[c] + FMIndex[i][c]
```

- Predecessor of index 1

```
c = BWT[1]
predec = 0['n'] + FMIndex[1]['n']
```

# 'n'  
# 5+0 = 5

- Predecessor of index 5

```
c = BWT[5]
predec = 0['a'] + FMIndex[5]['a']
```

# 'a'  
# 1+1 = 2

- Time to find predecessor:  $O(1)$

Index	Suffix Array	BWT	FM-index			
			\$	a	b	n
0	\$banana	a	0	0	0	0
1	a\$banan	n	0	1	0	0
2	ana\$ban	n	0	1	0	1
3	anana\$b	b	0	1	0	2
4	banana\$	\$	0	1	1	2
5	na\$bana	a	1	1	1	2
6	nana\$ba	a	1	2	1	2
7	<b>Counts</b>		1	3	1	2
	<b>Offset</b>		0	1	4	5

# Suffix Recovery



- What is the suffix array entry corresponding to BWT index  $i$ ?
  - Start at  $i$  and repeatedly find predecessors until  $i$  is reached again
- To find the *original* string, just start with  $i = 0$ , the '\$' index

```
In [39]: 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

          # recall that the FM-index that we built was "annb$aa", the BWT of "banana$"
          for i in range(len(bwt)):
              print(i, recoverSuffix(i, bwt, FM, Offset), bwt[i])
```

```
0 $banana a
1 a$banan n
2 ana$ban n
3 anana$b b
4 banana$ $
5 na$bana a
6 nana$ba a
```



# Finding Substrings

- Searches are performed in reverse order
- Searches return an interval of the suffix array that starts with the desired substring
  - Finds all occurrences of target
  - If there are no occurrences it finds an empty interval
- Starts with full BWT range (0, N)
- Narrows the range one symbol at a time
- To find substring "nana"

*# Initialize to full range of suffix array*

`lo, hi = 0, len(BWT)` *# len(BWT) = 7*

*# Find occurrences of "a"*

`lo = Offset['a'] + FMIndex[lo]['a']` *# lo = 1 + 0 = 1*

`hi = Offset['a'] + FMIndex[hi]['a']` *# hi = 1 + 3 = 4*

*# Find occurrences of "na"*

`lo = Offset['n'] + FMIndex[lo]['n']` *# lo = 5 + 0 = 5*

`hi = Offset['n'] + FMIndex[hi]['n']` *# hi = 5 + 2 = 7*

*# Find occurrences of "ana"*

`lo = Offset['a'] + FMIndex[lo]['a']` *# lo = 1 + 1 = 2*

`hi = Offset['a'] + FMIndex[hi]['a']` *# hi = 1 + 3 = 4*

*# Find occurrences of "nana"*

`lo = Offset['n'] + FMIndex[lo]['n']` *# lo = 5 + 1 = 6*

`hi = Offset['n'] + FMIndex[hi]['n']` *# hi = 5 + 2 = 7*

FM-index

Index	Suffix Array	BWT	FM-index			
			\$	a	b	n
0	\$banana	a	0	0	0	0
1	a\$banan	n	0	1	0	0
2	ana\$ban	n	0	1	0	1
3	anana\$b	b	0	1	0	2
4	banana\$	\$	0	1	1	2
5	na\$bana	a	1	1	1	2
6	nana\$b	a	1	2	1	2
7	<b>Counts</b>		1	3	1	2
	<b>Offset</b>		0	1	4	5

# In Python



One of the simplest, fastest, methods we've seen for searching

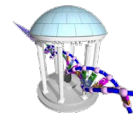
```
In [40]: def findBWT(pattern, FMIndex, Offset):
         lo = 0
         hi = len(FMIndex) - 1
         for symbol in reversed(pattern):
             lo = Offset[symbol] + FMIndex[lo][symbol]
             hi = Offset[symbol] + FMIndex[hi][symbol]
         return lo, hi

         print(findBWT("ana", FM, Offset))
         print(findBWT("ban", FM, Offset))
         print(findBWT("ann", FM, Offset))
```

```
(2, 4)
(4, 5)
(4, 4)
```

```
0 $banana
1 a$banan
2 ana$ban
3 anana$b
4 banana$
5 na$bana
6 nana$ba
```

# BWT score card



Method	Storage	Single Search	Multi Search
Brute Force	$O(m)$	$O(nm)$	$O(p \ n \ m)$
Keyword Tries	$O(pn)$	$O(nm)$	$O(p \ m)$
Suffix Trees	$O(m)^*$	$O(n)$	$O(p \ n)$
Suffix Arrays	$O(m)$	$O(n \log(m))$	$O(p \ n \ \log(m))$
BWT	$O(m)^\dagger$	$O(n)$	$O(p \ n)$

Where:

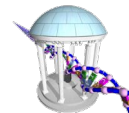
- $m$  is the length of the text to be searched
- $n$  is the length of the pattern (maximum length if more than 1)
- $p$  is the number of patterns

\* With large constants, however

† Usually significantly smaller than  $m$



# BWT Gotchas

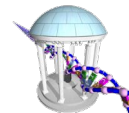


- While the BWT itself is small, its FM-index can be large
- A full FM-index requires  $O(|\Sigma| m)$  space
- But it can be sampled with minimal performance impact
  - rather than store the FM-index for all indices store only 1 in F
  - when accessing find the closest smaller instantiated index and use the BWT to fill in the requested missing values
- Example with  $F = 3$ 
  - when `FMIndex[5]['b']` is accessed
  - retrieve `FMIndex[3]['b'] = 0`
  - scan BWT from `[3:5]` counting 'b's (1) and adding them to the count at `FMIndex[3]`
  - return the count = 1
- In practice  $F$  values as large as 1000 have little performance impact
  - Why? BWT is small and tends to stay in cache
  - BWT is compressed so scanning through 1000 characters involves fewer reads
- To have all the capabilities of a Suffix Tree, a BWT needs an LCP array

Sampled FM-index

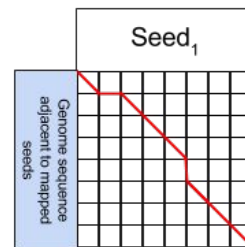
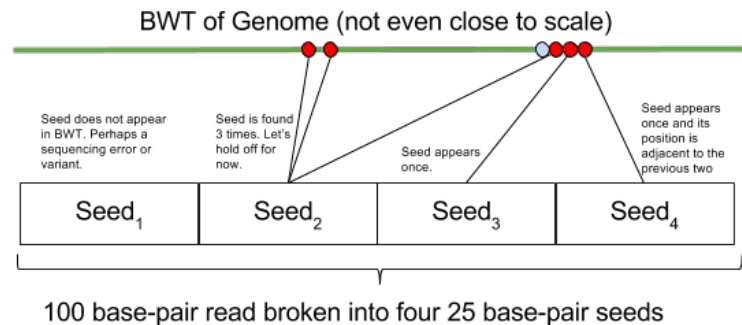
Index	Suffix Array	BWT	Sampled FM-index			
			\$	a	b	n
0	\$banana	a	0	0	0	0
1	a\$banan	n	0	1	0	0
2	ana\$ban	n	0	1	0	1
3	anana\$b	b	0	1	0	2
4	banana\$	\$	0	1	1	2
5	na\$bana	a	1	1	1	2
6	nana\$ba	a	1	2	1	2
7	<b>Counts</b>		1	3	1	2
	<b>Offset</b>		0	1	4	5

# Real-World uses of BWTs



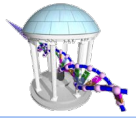
BWTs are the dominant representation and method used for **Sequence Alignment**

- **Sequence-Alignment Problem:** Given a collection of short nucleotide fragments (either DNA or RNA) find the best approximate alignment for each fragment in a reference genome
- Bowtie2 (2012) and BWA (2009) are the dominant aligners
- As a preprocess a BWT of the reference genome is built ( $\approx$  1-3 GB)
- How alignment works:
  - given a *read* from a sequenced fragment (72-150 base pairs typically)
  - cut the read into smaller seeds (25-31 base pairs typically)
  - Search for an exact match to each using the BWT
  - Use local alignment (dynamic program) to match the remaining bases



Perform a global sequence alignment on the unmatched seeds and report back the score

# Next Time



We go even deeper down the BWT rabbit hole

