Suffix Arrays and BWTs

A tweak to argsort()

- Recall argsort() from last time:

```
def argsort(input):
    return sorted(range(len(input)), cmp=lambda i,j: 1 if input[i] >= input[j] else -1)

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^{th} suffix starts at index i
  - thus we don't need to extract the suffixes, just use offsets

Constructing a Suffix Array

```
def suffixArray(text):
    return sorted(range(len(text)), cmp=lambda i,j: 1 if text[i:] >= text[j:] else -1)

t = "abananaban"
sa = suffixArray(t)
for i in sa:
    print "%2d: %s" % (i, t[i:])
```

```
 6: aban
 0: abananaban
 8: an
 4: anaban
 2: ananaban
 7: ban
 1: bananaban
 9: n
 5: naban
 3: nanaban
```

Searching a Suffix Array

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

```
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]:]
```

```
abananaban
2 8 an
```

Finding all Occurences

```
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)
for suffix in sa[first:last]:
    print suffix, t[suffix:]
```

```
abananaban
8  an
4  anaban
2  ananaban
```

Longest repeated substring?

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

```
def computeLCP(text, suffixarray):
    m = len(text)
    lcp = [0 for i in xrange(m)]
    for i in xrange(1,m):
        u = suffixarray[i-1]
        v = suffixarray[i]
        n = 0
        while text[u] == text[v]:
            n += 1
            u += 1
            v += 1
            if (u >= m) or (v >=m):
                break
        lcp[i] = n
    return lcp

lcp = computeLCP(t, sa)

print "SA,LCP,Suffix"
for i, j in enumerate(sa):
    print "%2d: %2d %s" % (j, lcp[i], t[j:])
```

- It should be evident that the longest repeated substring is at the suffix array index with the largest LCP
- It is shared with one or more suffixes before it

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 log(m))	O(n log(m))	O(p n log(m))

* with large constants, however

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$"
BWT(text) = "amnnn$lcpmnapaaaaaaala"
```

- 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 lexigraphically before any other symbol

Key Idea behind the BWT

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

"Cyclical Suffixes"	"Sorted Cyclical Suffixes"
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
$tarheel	tarheel$

- 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

BWT in Python

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

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

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

```
annb$aa
amnnn$lcpmnapaaaaaaaala
```

BWT from a Suffix Array

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

```
def BWTfromSuffixArray(text, suffixarray):
    return ''.join(text[i-1] for i in suffixarray)

newt =  t+'$'
sa = suffixArray(newt)
print newt
print sa
print BWTfromSuffixArray(newt, sa)
```

```
abananaban$
[10, 6, 0, 8, 4, 2, 7, 1, 9, 5, 3]
nn$bnbaaaaa
```

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 len(bwt) empty strings
2. repeat length(*bwt*) times:
3. prepend *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)

0	1	2	3	4	5	6	7	8
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

```
def inverseBWT(bwt):
    # initialize the table from t
    table = ['' for c in bwt]
    for j in xrange(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$")
print inverseBWT("amnnn$lcpmnapaaaaaaala")
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 tendancy 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 suffix array*** for pattern searching?

- In 2005 two researchers, Ferragina & Manzini, figured out how
- First, an important property they uncovered

"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 predecessor symbols of a suffix array preserve the relative suffix order
- The j^{th} occurance of a symbol in the BWT corresponds to its j^{th} occurance in the suffix array

$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba

- This property allows one two 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 occurences 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 $O(m|\Sigma|)$

			FM-index			
Index	**Suffix Array**	**BWT**	**$**	**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	**Counts**		1	3	1	2
	Offset		0	1	4	5

Python for constructing FM-index

```
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].iteritems()}
        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 "%2s,%2s,%2s,%2s" % tuple([symbol for symbol in sorted(Offset.keys())])
for row in FM:
    print "%2d,%2d,%2d,%2d" % tuple([row[symbol] for symbol in sorted(row.keys())])
```

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

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]                          # 'n'
predec = O['n'] + FMIndex[1]['n']   # 5+0 = 5
```

- Predecessor of index 5

```
c = BWT[5]                          # 'a'
predec = O['a'] + FMindex[5]['a']   # 1+1 = 2
```

- Time to find predecessor: O(1)

			FM-index			
Index	**Suffix Array**	**BWT**	**$**	**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	**Counts**		1	3	1	2
	Offset		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

```
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

print recoverSuffix(3, bwt, FM, Offset)
print recoverSuffix(0, bwt, FM, Offset)
```

```
anana$b
$banana
```

Finding Substrings

- Searches are performed in reverse order
- Searches return an interval of the suffix array that starts with the desired substring
  - Finds all occurences of target
  - If there are no occurences 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, 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**	**$**	**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	**Counts**		1	3	1	2
	Offset		0	1	4	5

In Python

- One of the simplest methods we've seen for searching

```
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)
```

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 log(m))	O(n log(m))	O(p n log(m))
BWT	O(m)†	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

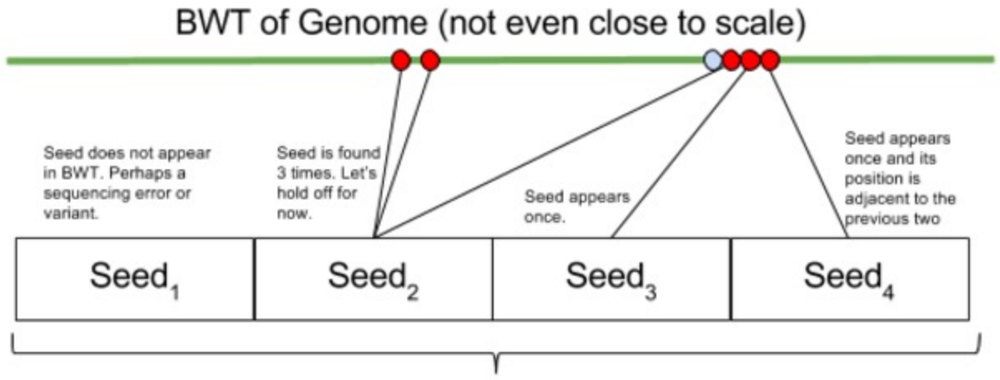
- While the BWT itself is small, its FM-index can be large
- A full FM-index requires *O(|Σ| m)* space
- But it can be sampled with minimal performance impact
  - rather than store the FM-index for all indices store only 1 in F
  - when accessing find the closest smaller instantiated index and use the BWT to fill in the requested missing values
- Example with F = 3
  - when FMIndex[5]['b'] is accessed
  - retrive 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
- To have all the capabilities of a Suffix Tree, a BWT needs an LCP array

			Sampled FM-index			
Index	**Suffix Array**	**BWT**	**$**	**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	**Counts**		1	3	1	2
	Offset		0	1	4	5

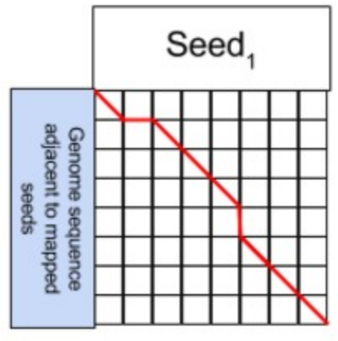
Real-World uses of BWTs

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 read in a reference genome
- Bowtie (2009) and BWA (2009) are the dominant aligners
- As a preprocess a BWT of the reference genome is built (≈ 1-3 GB)
- Alignment:
  - 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



100 base-pair read broken into four 25 base-pair seeds



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

Next Time

We go deeper down the BWT rabbit hole