Comp 555 - BioAlgorithms - Spring 2022

- Problem set #1 has been graded
- Problem set #2 is due on Thursday. Submission portal is open
- Problem set #3 will go out soon afterward.

Suffix Arrays and BWTs

A tweak to argsort()

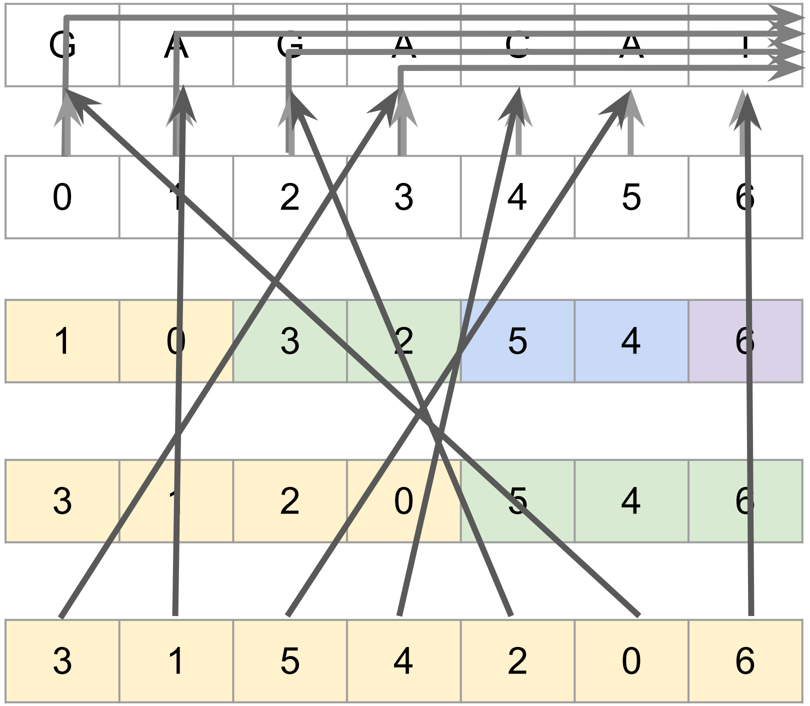
- Recall argsort() from last time:

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

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

```
[3, 1, 5, 4, 2, 6, 0]
```

- If we know that our input is suffixes from a single string
  - the ith suffix starts at index i
  - thus we don't need to extract the suffixes, just use offsets

Comparing Suffixes in Place

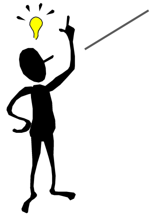


Original Order

Sort adjacent suffixes

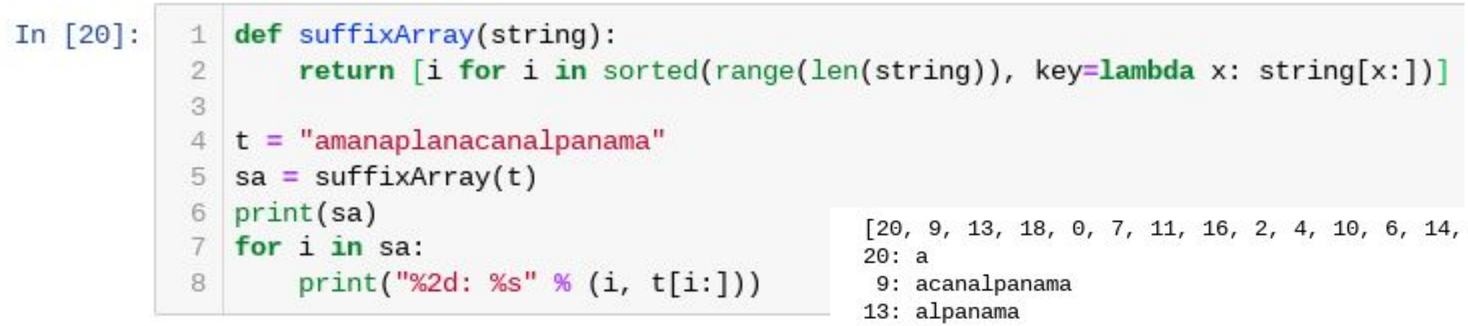
Merge sorted pairs

Merge sorted halves

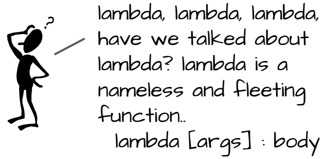


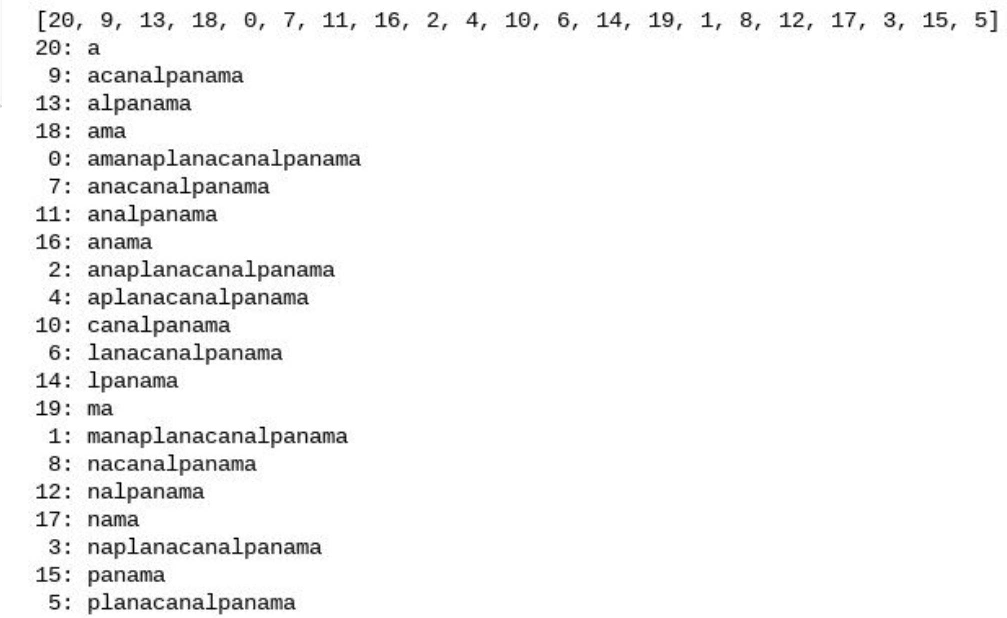
The key idea here is that we don't need to explicitly create a list of suffixes, since they are part of the original text.

Constructing a Suffix Array



```
In [20]: def suffixArray(string):
             return [i for i in sorted(range(len(string)), key=lambda x: string[x:])]

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


lambda, lambda, lambda, have we talked about lambda? lambda is a nameless and fleeting function..
lambda [args] : body

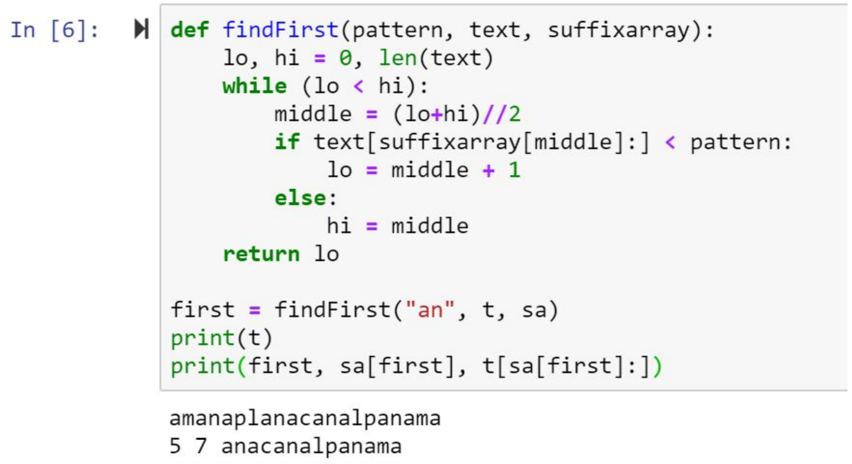


```
[20, 9, 13, 18, 0, 7, 11, 16, 2, 4, 10, 6, 14, 19, 1, 8, 12, 17, 3, 15, 5]
20: a
 9: acanalpanama
13: alpanama
18: ama
 0: amanaplanacanalpanama
 7: anacanalpanama
11: analpanama
16: anama
 2: anaplanacanalpanama
 4: aplanacanalpanama
10: canalpanama
 6: lanacanalpanama
14: lpanama
19: ma
 1: manaplanacanalpanama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

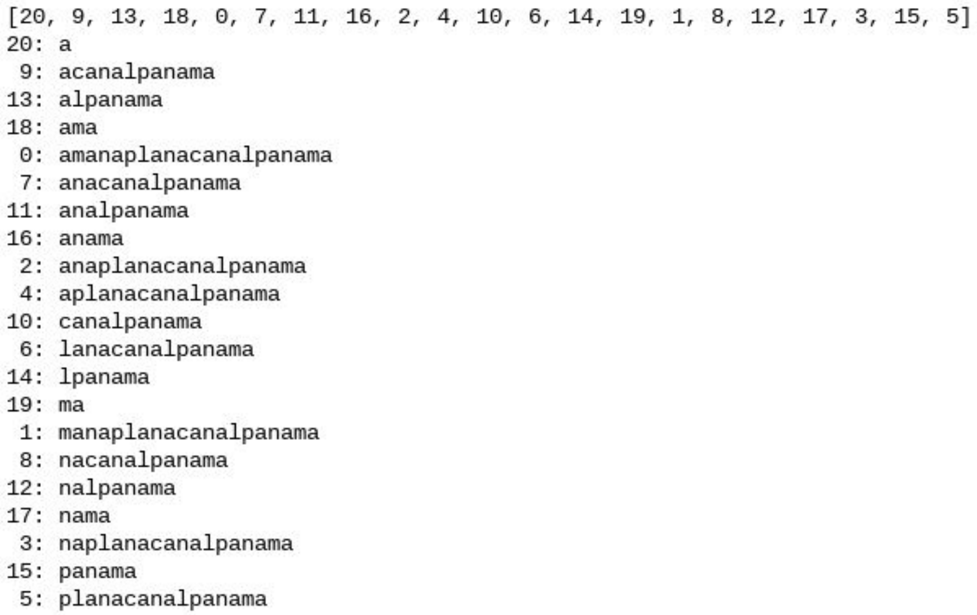
The "key" parameter defines a *lambda* function, which when given a value, x, from the given list, the range in this instance, determines what should be used for the comparison test of the sort. In this case it is a suffix from "string".

Searching a Suffix Array

- Searching a sorted list requires O(log(m)) comparisons using ***binary search***
- Each comparison requires examining at most *n* symbols of the pattern
- Thus, searching is O(nlog(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
```



```
[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
```

Finding all Occurrences

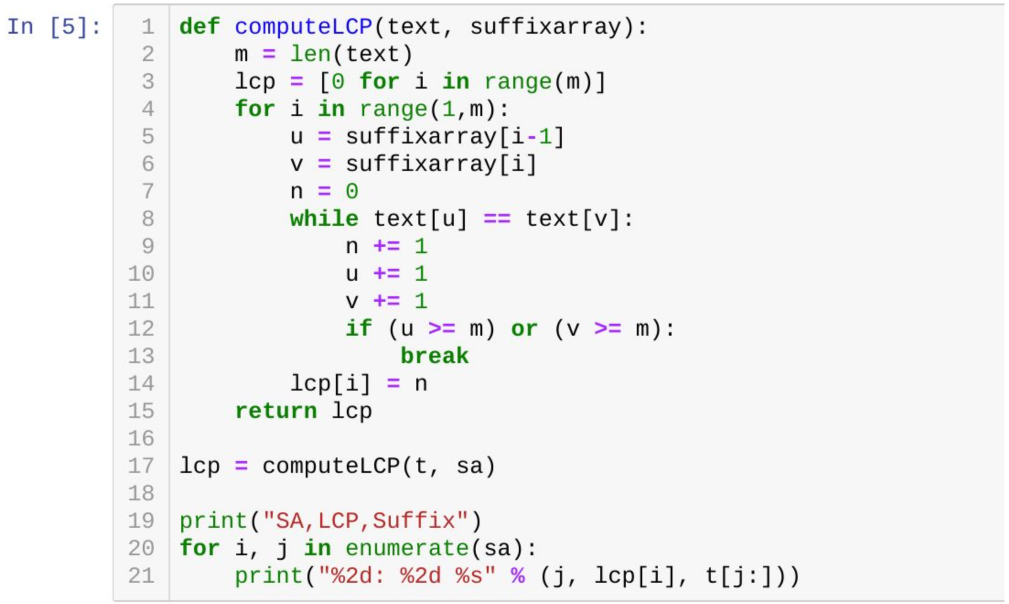
A variant to binary search which finds the last occurence of a pattern rather than the first. Only difference, uses "<=" instead of "<", but needs to trim string comparison to test for equality.

```
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* data structure, called the **Longest Common Prefix**, LCP



```
In [5]:
def computeLCP(text, suffixarray):
    m = len(text)
    lcp = [0 for i in range(m)]
    for i in range(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:]))
```

```
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 of Pattern Searching 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 Text 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 the text exhibits redundancy.

Example:

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

- Notice how the transformed text has long *runs* of repeated characters, 3 ns, 7 as
- 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:

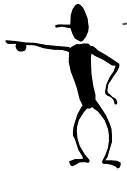
```
Compress(BWT(text)) = am3n$lcpmnap7ala  (16 chars instead of 22)
```

- 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

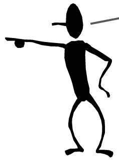
Key Idea behind the BWT

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

```
"Cyclical Suffixes"      "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
      $tarheel                  tarheel$
```



Cyclical Suffixes are formed by concatenating each **suffix** with its complementary **prefix** separated by a "end-of-text" character. In this case '$'. A nice property of cyclical suffixes is that they are all the same length.



The bwt is the concatenated predecessors

- The BWT of "tarheels" is the last column of the sorted cyclical suffixes "ltherea$"
- Notice that the sorted cyclical suffixes have a lot in common with a suffix array.
- The BWT is the "predecessors of all sorted cyclical suffixes", where the end-of-text symbol "$" is considered the first lexicographically

In [7]:
```python
# making cyclical suffixes
t="carolina$"
print([t[i:]+t[:i] for i in range(len(t))])
```

```
['carolina$', 'arolina$c', 'rolina$ca', 'olina$car', 'lina$caro', 'ina$carol', 'na$caroli', 'a$carolin', '$carolina']
```

BWT in Python

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

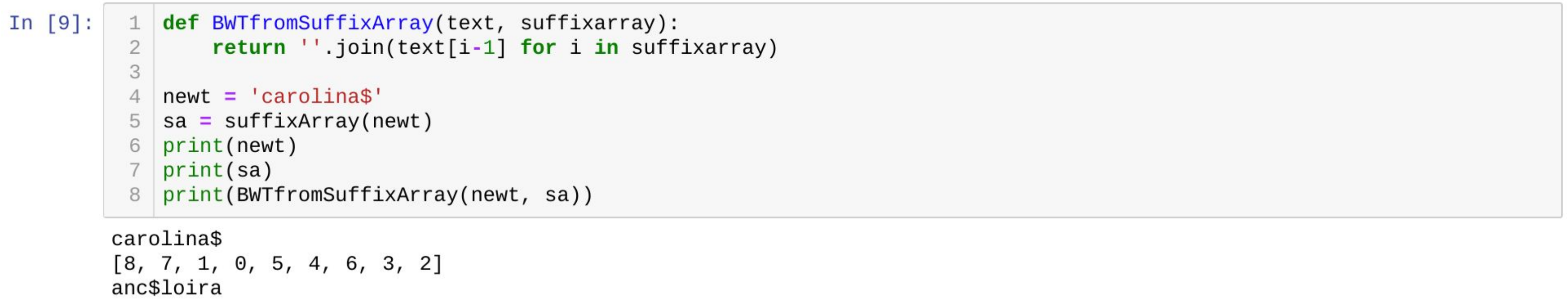
```
In [28]: def BWT(t):
             # create a sorted list of all cyclic suffixes of t
             rotation = sorted([t[i:]+t[:i] for i in range(len(t))])
             # concatenate the last symbols from each suffix
             return ''.join(r[-1] for r in rotation)

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

```
annb$aa
amnnn$lcpmnapaaaaaaala
nn$bnbaaaaa
```

BWT from a Suffix Array

- It is even simpler to compute the BWT from a Suffix Array
- Key idea: The BWT is the "predecessor" symbol of each sorted suffix

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



```
In [9]:
def BWTfromSuffixArray(text, suffixarray):
    return ''.join(text[i-1] for i in suffixarray)

newt = 'carolina$'
sa = suffixArray(newt)
print(newt)
print(sa)
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):

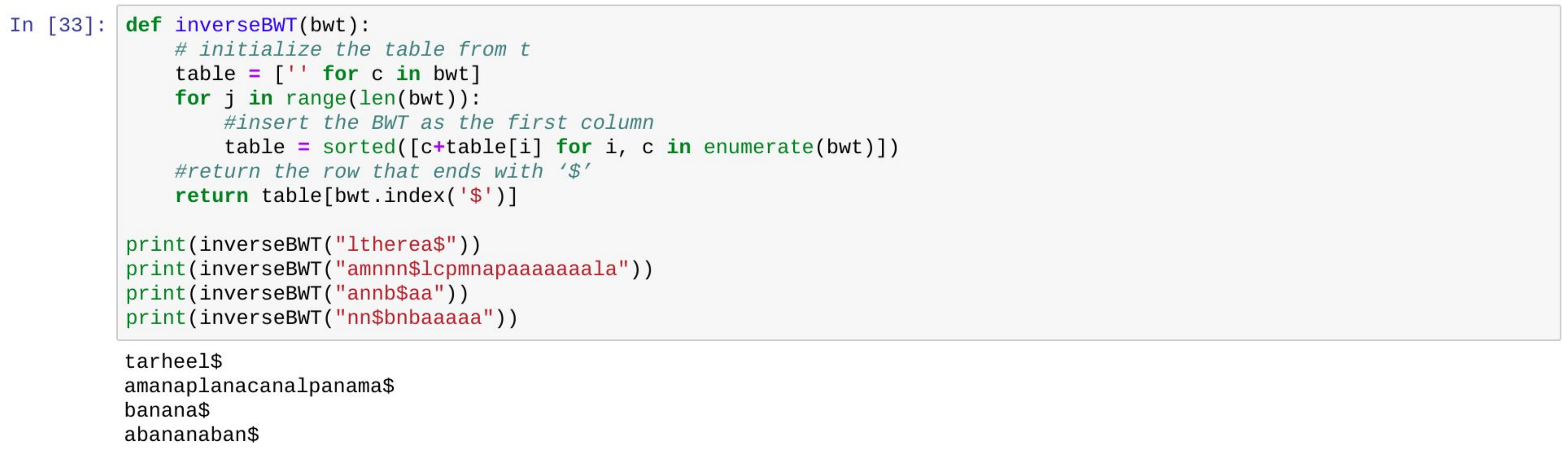
```
1.  Create a table of len(bwt) empty strings
2.  repeat length(bwt) times:
3.      prepend bwt as the first column of the table
4.      sort rows of the table alphabetically
5.  return (row of table with bwt's 'EOF' character)
```

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	$tarheel	tarheel$

- Or you could always return the first row, with the '$' first rather than last

Inverse BWT in Python

- Again, straightforward, but not the most efficient approach

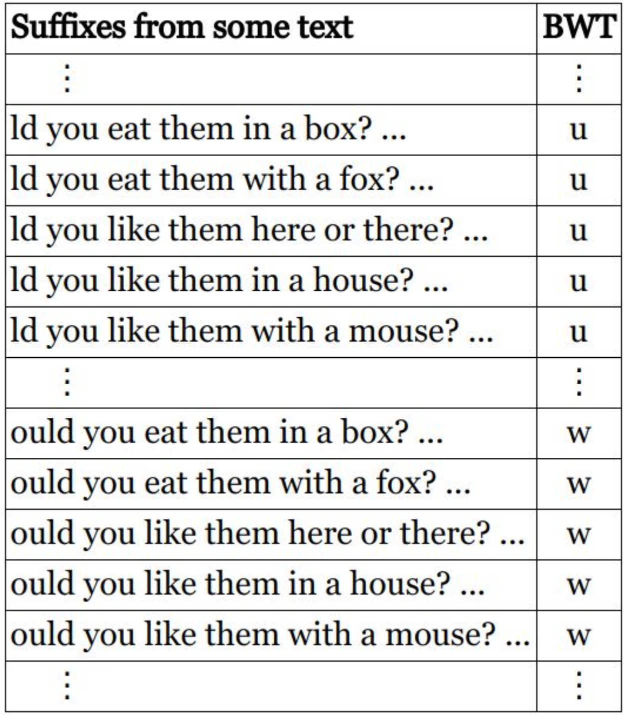


```
In [33]: def inverseBWT(bwt):
             # initialize the table from t
             table = ['' for c in bwt]
             for j in range(len(bwt)):
                 #insert the BWT as the first column
                 table = sorted([c+table[i] for i, c in enumerate(bwt)])
             #return the row that ends with ‘$’
             return table[bwt.index('$')]

         print(inverseBWT("ltherea$"))
         print(inverseBWT("amnnn$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 tendency to form long runs of repeated symbols
- Why does it form runs?
- Because, 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

"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 of the BWT

- The BWT transforms "banana$" to "annb$aa"
- The predecessor symbols of a suffix array preserve the relative suffix order
- The *jth occurrence of a symbol in the BWT corresponds to its jth 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$ba	3rd ‘a’ in BWT, 1st ‘a’ in banana, 3rd ‘a’ in suffix array

- 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 occurrences in the BWT matches the suffix array (recall it is a permutation)

Revisiting the First-Last (FL) mapping property

```
 i "hidden"suffix
 0 $annabananaban
 1 aban$annabanan
 2 abananaban$ann
 3 an$annabananab
 4 anaban$annaban
 5 ananaban$annab
 6 annabananaban$
 7 ban$annabanana
 8 bananaban$anna
 9 n$annabananaba
10 naban$annabana
11 nabananaban$an
12 nanaban$annaba
13 nnabananaban$a
```

The BWT for the string "annabananaban$" is "nnnbnb$aaaanaa". Last column on the left.

To extract the suffix with index 3.

The BWT tells us that the predecessor at 3 is "b".

FL tells us this is the 1st "b" in the BWT and thus is the first letter of the suffix with index 7. ("b")

The BWT tells us that the predecessor at 7 is "a".

FL tells us this is the 1st "a" in the BWT and thus is the first letter of the suffix with index 1. ("ab")

Revisiting the First-Last (F-L) mapping property

```
 i "hidden"suffix
 0 $annabananaban
 1 aban$annabanan
 2 abananaban$ann
 3 an$annabananab
 4 anaban$annaban
 5 ananaban$annab
 6 annabananaban$
 7 ban$annabanana
 8 bananaban$anna
 9 n$annabananaba
10 naban$annabana
11 nabananaban$an
12 nanaban$annaba
13 nnabananaban$a
```

The BWT tells us that the predecessor at 1 is "n".

FL tells us this is the 2nd "n" in the BWT and thus is the first letter of the suffix with index 10. ("nab")

The BWT tells us that the predecessor at 10 is "a".

FL tells us this is the 4th "a" in the BWT and thus is the first letter of the suffix with index 4. ("anab")

The BWT tells us that the predecessor at 4 is "n".

FL tells us this is the 4th "n" in the BWT and thus is the first letter of the suffix with index 12. ("nanab")

The BWT tells us that the predecessor at 12 is "a".

Revisiting the First-Last (F-L) mapping property

```
 i "hidden"suffix
 0 $annabananaban
 1 aban$annabanan
 2 abananaban$ann
 3 an$annabananab
 4 anaban$annaban
 5 ananaban$annab
 6 annabananaban$
 7 ban$annabanana
 8 bananaban$anna
 9 n$annabananaba
10 naban$annabana
11 nabananaban$an
12 nanaban$annaba
13 nnabananaban$a
```

The BWT tells us that the predecessor at 12 is "a".

FL tells us this is the 5th "a" in the BWT and thus is the first letter of the suffix with index 5. ("ananab")

The BWT tells us that the predecessor at 5 is "b".

FL tells us this is the 2nd "b" in the BWT and thus is the first letter of the suffix with index 8. ("bananab")

The BWT tells us that the predecessor at 8 is "a".

FL tells us this is the 2nd "a" in the BWT and thus is the first letter of the suffix with index 2. ("abananab")

The BWT tells us that the predecessor at 2 is "n".

FM-index keeps track of the F-L details

```
 i     suffix       bwt  $, a, b, n
 0 $annabananaban   n:   0, 0, 0, 0
 1 aban$annabanan   n:   0, 0, 0, 1
 2 abananaban$ann   n:   0, 0, 0, 2
 3 an$annabananab   b:   0, 0, 0, 3
 4 anaban$annaban   n:   0, 0, 1, 3
 5 ananaban$annab   b:   0, 0, 1, 4
 6 annabananaban$   $:   0, 0, 2, 4
 7 ban$annabanana   a:   1, 0, 2, 4
 8 bananaban$anna   a:   1, 1, 2, 4
 9 n$annabananaba   a:   1, 2, 2, 4
10 naban$annabana   a:   1, 3, 2, 4
11 nabananaban$an   n:   1, 4, 2, 4
12 nanaban$annaba   a:   1, 4, 2, 5
13 nnabananaban$a   a:   1, 5, 2, 5
                         1, 6, 2, 5
```

The FM-index does the F-L mapping.

The final symbol counts provide the suffix offsets:
'$': 0, 'a':0+1 = 1, 'b':1+6 =7, 'n':7+2=9

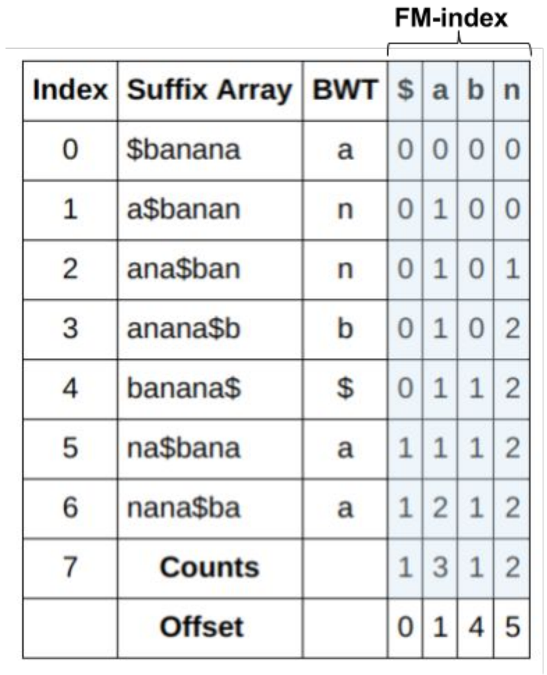
For example:
The BWT at 5 is "b".

The FM-index at 5 gives us a 1 for 'b' which is added to the 'b' suffix offset, 7, to get the suffix 8.

Implementing "F-L tells us this is the 2nd "b" in the BWT and thus is the first letter of the suffix with index 8."

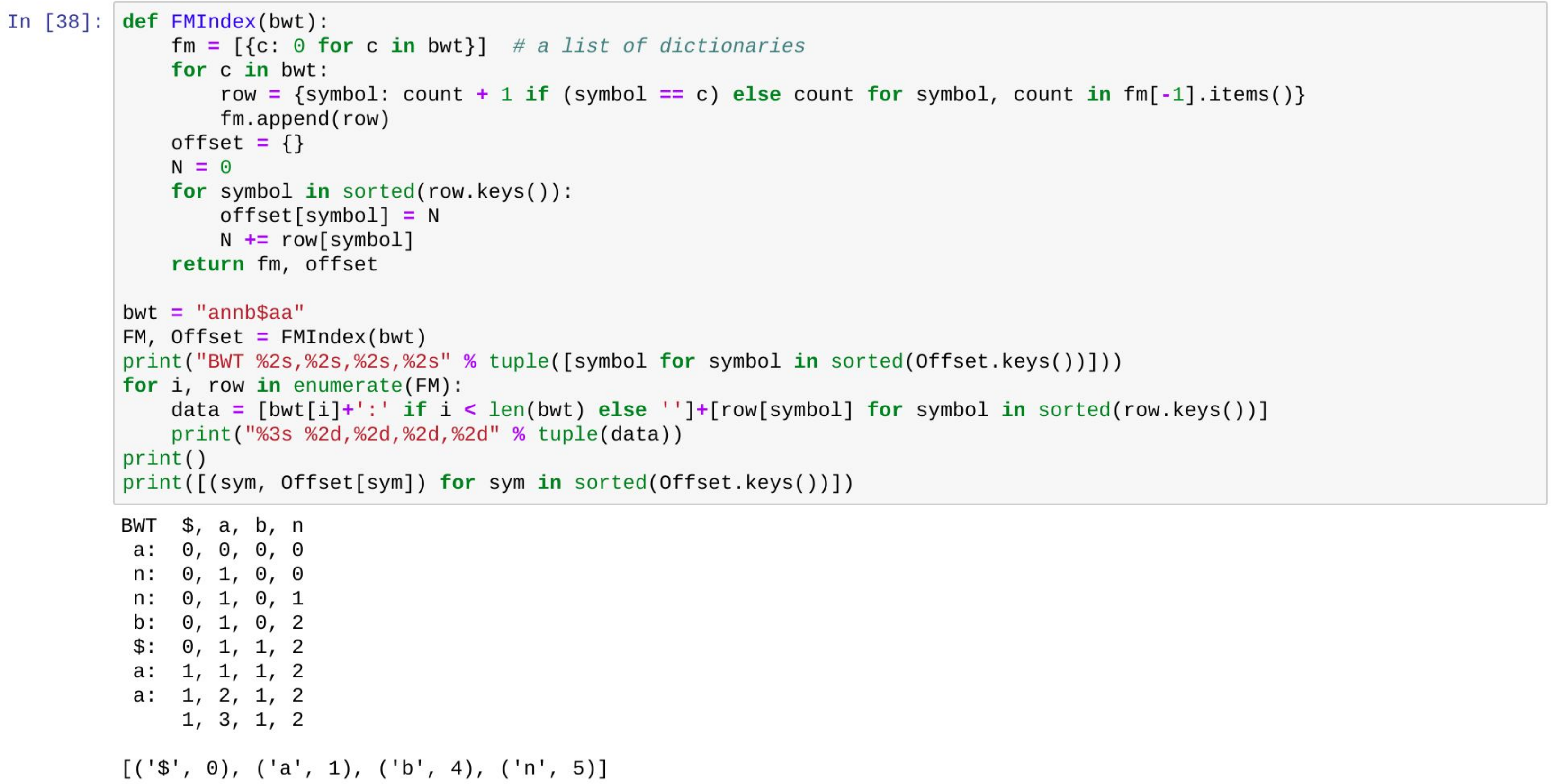
The FM-index

- The FM-index is another *helper* data structure like the *LCP array* mentioned previously
- It is a 2D array whose size is [|*text*|+1,|Σ|], where |Σ| is the alphabet size
- It keeps track of how many of each symbol have been seen in the BWT prior to its ith 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*|Σ|)



			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

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]
```

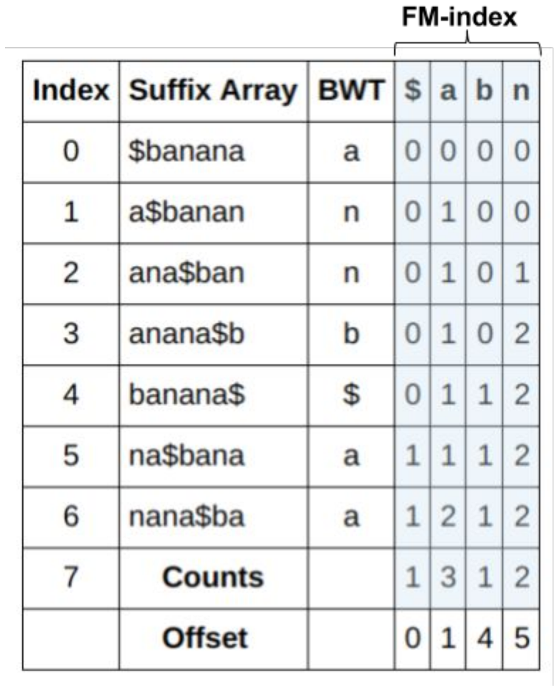
- Predecessor of index 1

```
c = BWT[1]                                # '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)



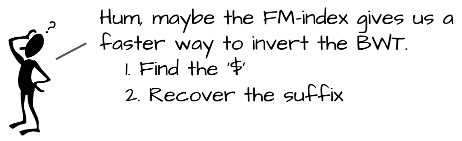
			FM-index			
Index	**Suffix Array**	**BWT**	**$**	**a**	**b**	**n**
0	$banana	a	0	0	0	0
1	a$banan	n	0	1	0	0
2	ana$ban	n	0	1	0	1
3	anana$b	b	0	1	0	2
4	banana$	$	0	1	1	2
5	na$bana	a	1	1	1	2
6	nana$ba	a	1	2	1	2
7	**Counts**		1	3	1	2
	Offset		0	1	4	5

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

```
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
```

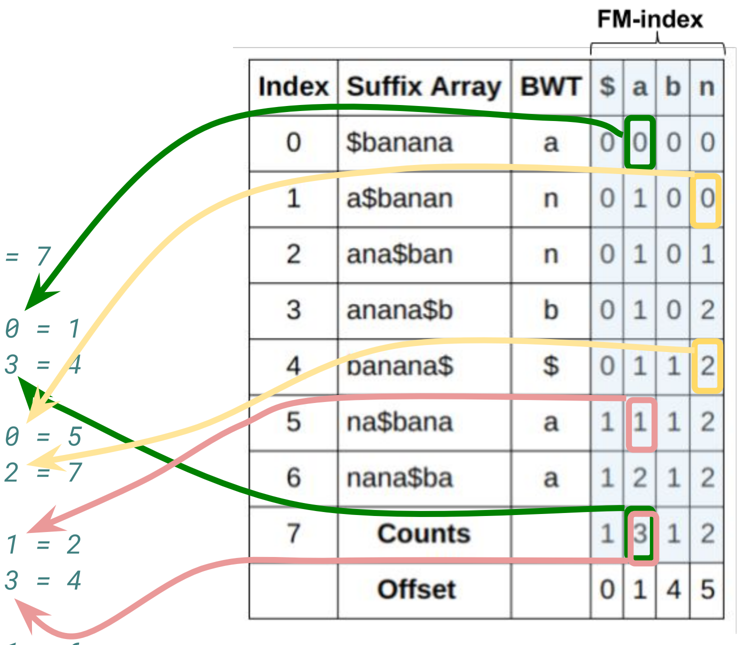

Hum, maybe the FM-index gives us a faster way to invert the BWT.
1. Find the '$'
2. Recover the suffix

			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

Finding Patterns using a BWT

- Searches are performed in reverse order
- Searches return an interval of the suffix array that starts with the desired substring
  - Finds all occurrences of pattern
  - 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
```



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

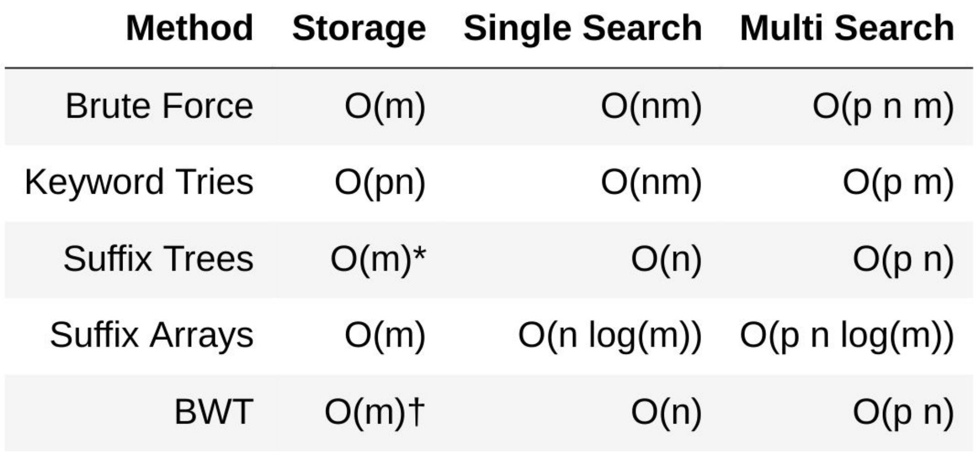
One of the simplest, fastest, methods we've seen for searching

```
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)†	O(n)	O(p n)

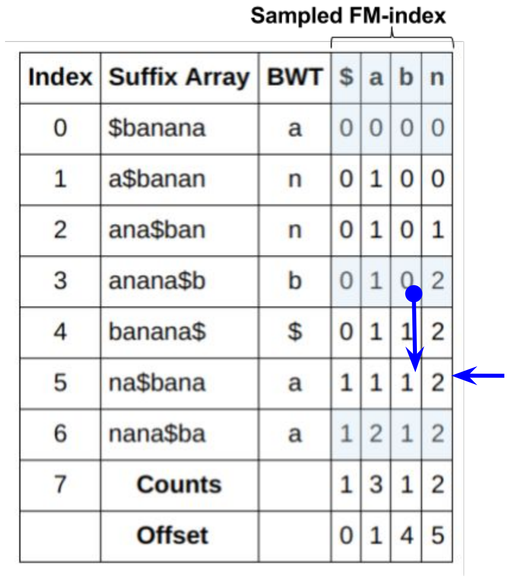
Where:

- *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
  - 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



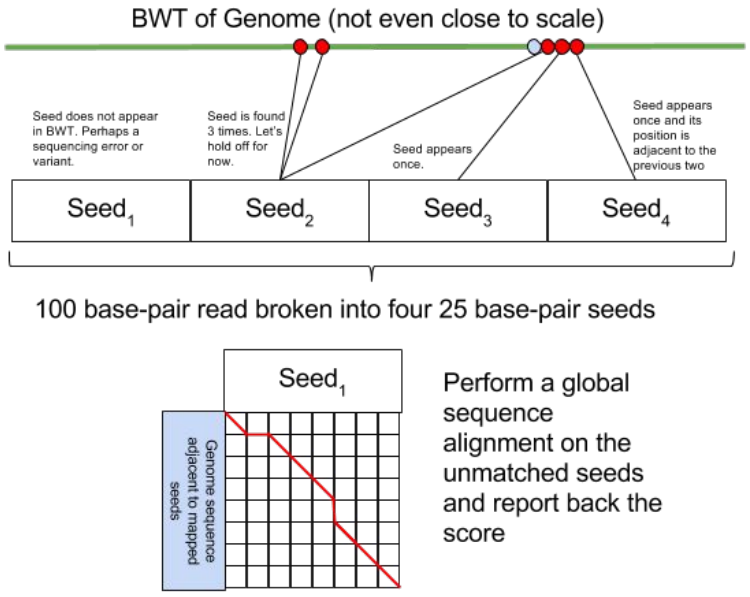
			Sampled FM-index			
Index	Suffix Array	BWT	$	a	b	n
0	$banana	a	0	0	0	0
1	a$banan	n	0	1	0	0
2	ana$ban	n	0	1	0	1
3	anana$b	b	0	1	0	2
4	banana$	$	0	1	1	2
5	na$bana	a	1	1	1	2
6	nana$ba	a	1	2	1	2
7	Counts		1	3	1	2
	Offset		0	1	4	5

Real-World uses of BWTs

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

- <u>Sequence-Alignment Problem:</u> Given a collection of short nucleotide read (either DNA or RNA) find the best approximate alignment for each read in a reference genome

- Bowtie2 (2012) and BWA (2009) are the dominant aligners
- As a preprocess a BWT of the reference genome is built (≈ 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 genome BWT
  - Use local alignment (dynamic program) to match the remaining bases


BWT of Genome (not even close to scale)
Seed does not appear in BWT. Perhaps a sequencing error or variant.
Seed is found 3 times. Let's hold off for now.
Seed appears once.
Seed appears once and its position is adjacent to the previous two
Seed_1
Seed_2
Seed_3
Seed_4
100 base-pair read broken into four 25 base-pair seeds
Seed_1
Genome sequence adjacent to mapped seeds
Perform a global sequence alignment on the unmatched seeds and report back the score

Time for an Exercise

Go to the course website and do the exercise linked to your setup page

"If you haven't exercised in a while, you may need to stretch and warm up before you stretch and warm up."

Next Time

We go even deeper down the BWT rabbit hole