BWT Alignment Notes

Michael Schatz (mschatz@cshl.edu)

==

Motivation:
  Searching for occurrences of a query string in a reference string is an
  extremely common computation, and forms the basis for genotyping, RNA-seq
  expression, ChIP-seq peak finding, WGA, BLAST, motif finding, etc, etc, etc

1. Exact Matching

==

```
G=Genome        n=Genome length
Q=Query         l=Query Length

      Typically n >> l

Brute force matching:
  Trivial to implement
  Extremely slow: O(n*l) naive or O(n+l) smart
  Space efficient: (O(n+l)) 3 billion bytes for 3Gbp genome
```

2. Suffix Arrays and Binary search

==

Brute force is slow because we check locations that cant possibly be a match
Need to skip or focus on portions of the genome likely to contain a match
using an index!

Phone Book analogy
  Play hi-low game to look up schatz in the phone book
  In lg(n) lookups, will zoom in on schatz.
  All occurrences will be next to each other
  If there are no occurrences, you can quit without fear of missing

Suffix array as full text index of the genome:
   allows searching for queries of any length at any position

Example

```
  G=GATTACA

  Suffixes             Sorted Suffixes

  0 GATTACA              6 A
  1  ATTACA              4 ACA
  2   TTACA              1 ATTACA
  3    TACA              5 CA
  4     ACA              0 GATTACA
  5      CA              3 TACA
  6       A              2 TTACA

  SA = 6,4,1,5,0,3,2
```

Can't explicitly store all suffixes or it would require O(n^2) space!

```
Suffix Array Search
  Binary Search:
       O(l lg n); can be reduced to O(lg n) by storing LCP array
  Space:
       N integers (offsets) + N bytes (string)
       15 billion bytes for 3 Gbp genome
  Constructing SA:
       Naive O(n^2 lg n), fast: O(n).
       Run once "overnight", amortize cost for many queries
```

3. Burrows-Wheeler Transform

==

Want compact space O(n) bytes *and* efficient search O(lg n) or O(l)
Goal: Optimal space index is 1 byte index per byte of text (full text index)

BWT has these properties, plus other cool properties.

Named for Michael Burrow and David Wheeler while working at DEC in 1994
Original algorithm by Wheeler in 1983
Currently one of the most popular index structures for genomic searches:
  Bowtie/Bowtie2/TopHat, BWA, SOAP2, BLASR, ...

3.1 Construction / Definition

==

  Sort all cyclic rotations of G'=G$ where G is genome and $ is EOF character
  that is lexicographically less than all other characters in G

```
  Example:
    G=GATTACA
    G'=GATTACA$

   Rotations:             Sorted  (also called BWM)

    GATTACA$                   $GATTACA
    ATTACA$G                   A$GATTAC
    TTACA$GA                   ACA$GATT
    TACA$GAT                   ATTACA$G
    ACA$GATT                   CA$GATTA
    CA$GATTA                   GATTACA$
    A$GATTAC                   TACA$GAT
    $GATTACA                   TTACA$GA

   BWT (last column of BWM)          -^

     ACTGA$TA
```

That's it, no other tables needed. Not obvious here, but the BWT implicitly
encodes the suffix array. Sorting in this way also tends to cluster characters
together making it easier to compress -- this was the original motivation
for it. Also the key insight for the common bzip2 compression alg.

3.2 Last-first property

==

The magic of the BWT is the LF property: The ith occurrence of character C in
the last column *is* the ith occurrence of character C in the first column.

Why is this?
  Lets consider a schematic diagram of the BWM of a DNA string

```
    $ _ _ _ _ _ _ _   <- By construction, first row starts with $
    A _ _ _ _ _ _ _
    A _ _ _ _ _ _ _   <- Followed by section for A
    A _ _ _ _ _ _ _
    ...
    C _ _ _ _ _ _ _
    C _ _ _ _ _ _ _   <- Followed by C
    C _ _ _ _ _ _ _
    ...
    G _ _ _ _ _ _ _
    G _ _ _ _ _ _ _   <- Followed by G
    G _ _ _ _ _ _ _
    ...
    T _ _ _ _ _ _ _
    T _ _ _ _ _ _ _   <- Followed by T
    T _ _ _ _ _ _ _
```

Lets call those three rotations that start with C rotations X, Y, and Z
The first character of each of those rotations is x, y, z (without loss
of generality -- we don't know what those strings are, but we can label
the characters)

```
  ...
  C x X X X X X X
  C y Y Y Y Y Y Y
  C z Z Z Z Z Z Z
  ...
```

Now since the BWM contains *every* cyclic rotation, we know those 3 C strings
will also be rotated like so, someplace else in the BWM

```
  CxXXXXXX       xXXXXXXC
  CyYYYYYY  =>   yYYYYYYC
  CzZZZZZZ       zZZZZZZC
```

Key insight: Since the rotations are sorted, we know that X < Y < Z
             and x <= y <= z. As such their relative placement must also
             be in sorted order in the BWM when C is rotated to the last
             column.

```
  $ _ _ _ _ _ _ _
  A _ _ _ _ _ _ _
  A X X X X X X C   <- Possible location of X (x=A)
  A _ _ _ _ _ _ _
  ...
  C x X X X X X X
  C y Y Y Y Y Y Y   <- Original locations of X, Y, Z
  C z Z Z Z Z Z Z
  ...
  G Y Y Y Y Y Y Y   <- Possible location of Y (must be below X, y=G)
  G _ _ _ _ _ _ _
  G _ _ _ _ _ _ _
  ...
  T _ _ _ _ _ _ _
  T _ _ _ _ _ _ _
  T Z Z Z Z Z Z C   <- Possible location of Z (must be below Y, z=T)
```

Last-First property is actually a statement of the *rest* of the rotation.
When they are sorted as the second character of the rotation, they are also
sorted when they are the first character of the rotation so the ranks must
be the same.

3.2 Unwinding the BWT
==

How can we use the LF-property to reconstruct G from BWT(G)?
Say the BWT is ACTTGA$TTAA (11 characters)

```
This means the genome must looks like _ _ _ _ _ _ _ _ _ _ $
                                      1 2 3 4 5 6 7 8 9 0 1
```

Since the BWT is a permutation of G, we actually know a lot about how
the BWM must look: 1x$, 4xA, 1xC, 1xG, 4xT

And the BWM must look like

```
1 2 3 4 5 6 7 8 9 0 1
$ _ _ _ _ _ _ _ _ _ A  <- By construction, $ is first
A _ _ _ _ _ _ _ _ _ C  <- Must have 4 A rows
A _ _ _ _ _ _ _ _ _ T     "
A _ _ _ _ _ _ _ _ _ T     "
A _ _ _ _ _ _ _ _ _ G     "
C _ _ _ _ _ _ _ _ _ A  <- 1 C row
G _ _ _ _ _ _ _ _ _ $  <- 1 G row
T _ _ _ _ _ _ _ _ _ T  <- 4 T rows
T _ _ _ _ _ _ _ _ _ T     "
T _ _ _ _ _ _ _ _ _ A     "
T _ _ _ _ _ _ _ _ _ A     "

                    ^- Last column defined by the BWT
```

Since, the first row starts with '$' and the last character in that row
is A, we know the last character of the genome is A.

```
                     _ _ _ _ _ _ _ _ _ A $
                     1 2 3 4 5 6 7 8 9 0 1
```

With this we know the last character is A. So what is the character that
comes before that A? There are 4 rows that start with A, so the character
must be one of C,T,T, or G, but which one is it? Here is where we can use
the LF property: the A in the last column of $...A is the first A, so this
corresponds to the first row with A. The BWM must be:

```
                     1 2 3 4 5 6 7 8 9 0 1
                     $ _ _ _ _ _ _ _ _ _ A  <- 1st A last column
  1st A in first ->  A $ _ _ _ _ _ _ _ _ C  <- Must precede that A
                     ...
```

Now we know the character before A$ must be C:

```
                     _ _ _ _ _ _ _ _ C A $
                     1 2 3 4 5 6 7 8 9 0 1
```

Now this row has the 1st C in the last column, so that must correspond to
the 1st C in the first column

```
                     1 2 3 4 5 6 7 8 9 0 1
                     $ _ _ _ _ _ _ _ _ _ A
                     A $ _ _ _ _ _ _ _ _ C  <- 1st C
                     A _ _ _ _ _ _ _ _ _ T
                     A _ _ _ _ _ _ _ _ _ T
                     A _ _ _ _ _ _ _ _ _ G
  1st C in first ->  C A $ _ _ _ _ _ _ _ A  <- must precede that A
                     ...
```

Now we know the genome must be:

```
                     _ _ _ _ _ _ _ A C A $
                     1 2 3 4 5 6 7 8 9 0 1
```

Use the LF again

```
                     1 2 3 4 5 6 7 8 9 0 1
                     $ _ _ _ _ _ _ _ _ _ A
                     A $ _ _ _ _ _ _ _ _ C
  2nd A in first ->  A C A $ _ _ _ _ _ _ T  <- preceded by T
                     A _ _ _ _ _ _ _ _ _ T
                     A _ _ _ _ _ _ _ _ _ G
                     C A $ _ _ _ _ _ _ _ A  <- 2nd A in last
                     ...
```

Now we know the genome must be:

```
                     _ _ _ _ _ _ T A C A $
                     1 2 3 4 5 6 7 8 9 0 1
```

Use the LF again:

```
                     1 2 3 4 5 6 7 8 9 0 1
                     $ _ _ _ _ _ _ _ _ _ A
                     A $ _ _ _ _ _ _ _ _ C
                     A C A $ _ _ _ _ _ _ T  <- 1st T in last
                     A _ _ _ _ _ _ _ _ _ T
                     A _ _ _ _ _ _ _ _ _ G
                     C A $ _ _ _ _ _ _ _ A
                     G _ _ _ _ _ _ _ _ _ $
  1st T in first  -> T A C A $ _ _ _ _ _ T  <- preceded by T
                     ...
```

Now we know the genome must be:

```
                    _ _ _ _ _ T T A C A $
                    1 2 3 4 5 6 7 8 9 0 1
```

Use the LF again

```
                    1 2 3 4 5 6 7 8 9 0 1
                    $ _ _ _ _ _ _ _ _ _ A
                    A $ _ _ _ _ _ _ _ _ C
                    A C A $ _ _ _ _ _ _ T
                    A _ _ _ _ _ _ _ _ _ T
                    A _ _ _ _ _ _ _ _ _ G
                    C A $ _ _ _ _ _ _ _ A
                    G _ _ _ _ _ _ _ _ _ $
                    T A C A $ _ _ _ _ _ T <- 3rd T in last
                    T _ _ _ _ _ _ _ _ _ T
  3rd T in first -> T T A C A $ _ _ _ _ A <- preceded by A
                    T _ _ _ _ _ _ _ _ _ A
```

Now we know the genome must be:

```
                    _ _ _ _ A T T A C A $
                    1 2 3 4 5 6 7 8 9 0 1
```

Use the LF again

```
                    1 2 3 4 5 6 7 8 9 0 1
                    $ _ _ _ _ _ _ _ _ _ A
                    A $ _ _ _ _ _ _ _ _ C
                    A C A $ _ _ _ _ _ _ T
  3rd A in first -> A T T A C A $ _ _ _ T <- preceded by T
                    A _ _ _ _ _ _ _ _ _ G
                    C A $ _ _ _ _ _ _ _ A
                    G _ _ _ _ _ _ _ _ _ $
                    T A C A $ _ _ _ _ _ T
                    T _ _ _ _ _ _ _ _ _ T
                    T T A C A $ _ _ _ _ A <- 3rd A in last
                    T _ _ _ _ _ _ _ _ _ A
```

Now we know the genome must be:

```
                    _ _ _ T A T T A C A $
                    1 2 3 4 5 6 7 8 9 0 1
```

Use the LF again

```
                    1 2 3 4 5 6 7 8 9 0 1
                    $ _ _ _ _ _ _ _ _ _ A
                    A $ _ _ _ _ _ _ _ _ C
                    A C A $ _ _ _ _ _ _ T
                    A T T A C A $ _ _ _ T <- 2nd T in last
                    A _ _ _ _ _ _ _ _ _ G
                    C A $ _ _ _ _ _ _ _ A
                    G _ _ _ _ _ _ _ _ _ $
                    T A C A $ _ _ _ _ _ T
  2nd T in first -> T A T T A C A $ _ _ T <- preceded by T
                    T T A C A $ _ _ _ _ A
                    T _ _ _ _ _ _ _ _ _ A
```

Now we know the genome must be:

```
                    _ _ T T A T T A C A $
                    1 2 3 4 5 6 7 8 9 0 1
```

Use the LF again

```
                    1 2 3 4 5 6 7 8 9 0 1
                    $ _ _ _ _ _ _ _ _ _ A
                    A $ _ _ _ _ _ _ _ _ C
                    A C A $ _ _ _ _ _ _ T
                    A T T A C A $ _ _ _ T
                    A _ _ _ _ _ _ _ _ _ G
                    C A $ _ _ _ _ _ _ _ A
                    G _ _ _ _ _ _ _ _ _ $
                    T A C A $ _ _ _ _ _ T
                    T A T T A C A $ _ _ T <- 4th T in last
                    T T A C A $ _ _ _ _ A
  4th T in first -> T T A T T A C A $ _ A <- preceded by A
```

Now we know the genome must be:

```
                    _ A T T A T T A C A $
                    1 2 3 4 5 6 7 8 9 0 1
```

Use the LF again

```
                    1 2 3 4 5 6 7 8 9 0 1
                    $ _ _ _ _ _ _ _ _ _ A
                    A $ _ _ _ _ _ _ _ _ C
                    A C A $ _ _ _ _ _ _ T
                    A T T A C A $ _ _ _ T
  4th A in first -> A T T A T T A C A $ G <- preceded by G
                    C A $ _ _ _ _ _ _ _ A
                    G _ _ _ _ _ _ _ _ _ $
                    T A C A $ _ _ _ _ _ T
                    T A T T A C A $ _ _ T
                    T T A C A $ _ _ _ _ A
                    T T A T T A C A $ _ A <- 4th A in last
```

Now we know the genome must be:

```
                    G A T T A T T A C A $
                    1 2 3 4 5 6 7 8 9 0 1
```

At this point we can stop because we have processed all 11 characters, or we could apply the LF rule again, jump to the first G, and recognize the last column had a $.

```
                    1 2 3 4 5 6 7 8 9 0 1
                    $ _ _ _ _ _ _ _ _ _ A
                    A $ _ _ _ _ _ _ _ _ C
                    A C A $ _ _ _ _ _ _ T
                    A T T A C A $ _ _ _ T
                    A T T A T T A C A $ G <- 1st G in last
                    C A $ _ _ _ _ _ _ _ A
  1st G in first -> G A T T A T T A C A $ <-- all done!
                    T A C A $ _ _ _ _ _ T
                    T A T T A C A $ _ _ T
                    T T A C A $ _ _ _ _ A
                    T T A T T A C A $ _ A
```

In this way we can UNWIND the BWT back to the original genome. If we didn't start UNWINDING from the first row, we could determine the prefix (offset) of any row in the BWT. (See below)

3.3 Exact Matching

Great, we can use LF to unwind the BWT back to the original genome. Amazingly we can using a variant of LF to rapidly compute exact matches. The variant of LF called LFc "pretends" that a given character is present at the end of a given row.

General points:
- 2 phases:
  1. Use LFc to find a range of rows in the BWM that exactly match, similar to how binary search identifies a range of rows
  2. For each row, UNWIND back to the beginning of the genome to find the genome location (as opposed to the SA offset)
- Scan the query string backwards from end to beginning using LFc l times
  1. Use a top pointer and bottom pointer to track current valid range
  2. We know the query does not exist if top >= bottom
  3. Basic algorithm only supports exact matches

Example: Find all occurrences of ATT in BWT of ACTTGA$TTAA
(The answer should be positions 2 and 5)

From the BWT we can count characters to write the first column. The rest of the matrix is hidden. Initialize top pointer to first row, and bottom pointer to just beyond last row, and pretend that character is a T since that is the last character of ATT

```
      top -> $...A  <- if this was a T it would be the 1st T
             A...C
             A...T
             A...T
             A...G
             C...A
             G...$
             T...T
             T...T
             T...A
             T...A
      bot ->        <- if this was a T it would be the 5th T
```

Apply the LFc to jump to the range between the 1st and 5th T

```
             $...A
             A...C
             A...T
             A...T
             A...G
             C...A
             G...$
      top -> T...T
             T...T
             T...A
             T...A
      bot ->
```

This defines that range of rows that all start with 'T'. Now apply LFc pretending the last character was 'T' (since this is the second T)

```
             $...A
             A...C
             A...T
             A...T
             A...G
             C...A
             G...$
      top -> T...T <- if this was a T it would be the 3rd T
             T...T
             T...A
             T...A
      bot ->       <- if this was a T it would be the 5th T
```

Apply LFc

```
             $...A
             A...C
             A...T
             A...T
             A...G
             C...A
             G...$
             T...T
             T...T
      top -> T...A
             T...A
      bot ->
```

This defines the range of rows that begin 'TT'. Apply LFc with A

```
             $...A
             A...C
             A...T
             A...T
             A...G
             C...A
             G...$
             T...T
             T...T
      top -> T...A <- If this was an A it would be the 3rd A
             T...A
      bot ->       <- If this was an A it would be the 5th A
```

Apply LFc

```
             $...A
             A...C
             A...T
      top -> A...T
             A...G
      bot -> C...A
             G...$
             T...T
             T...T
             T...A
             T...A
```

Success! We have processed all the query characters and top < bot so we have a valid range of rows [3,5). Apply UNWIND(3) and UNWIND(4) to find the locations in the original genome

UNWIND(3)

```
                2nd T      4th T     4th A     1st G

            $...A     $...A     $...A     $...A     $...A
            A...C     A...C     A...C     A...C     A...C
            A...T     A...T     A...T     A...T     A...T
   start -> A...T -   A...T     A...T     A...T     A...T
            A...G  |  A...G     A...G   - A...G -   A...G
            C...A  |  C...A     C...A  |  C...A  |  C...A
            G...$  |  G...$     G...$  |  G...$   - G...$  <- offset 5
            T...T  |  T...T     T...T  |  T...T     T...T
            T...T   - T...T -   T...T  |  T...T     T...T
            T...A     T...A  |  T...A  |  T...A     T...A
            T...A     T...A   - T...A -   T...A     T...A

           shift:  1         2         3         4
```

UNWIND(4) is just like starting at the 4th A.

3.4 FM-index and other Practical considerations
==

Unwinding all the way to the beginning is expensive: O(n) steps.
So, instead of going all the way to the beginning of the string, periodically
leave a "breadcrumb" so that we can quickly find our place. The FM-index
accomplished this by sampling the suffix array every 16th or 32nd row
which is enough to guarantee a constant number of UNWIND steps.

FM-index/BWT best suited for exact matches only. Searching for inexact
matches is tricky: use the exact match algorithm to find long exact
matches, but then backtrack, permute the "worst" base and try searching
again.

Today, Bowtie2/BWA/BLASR/SOAP2 use the FM-index to find exact alignment
seends, and then use dynammic programming around those seeds

4. Research Questions
==

1. Faster construction over large databases of strings
2. Faster searching with mismatches and/or on special hardware
3. Bi-directional BWT: Search forward or reverse
4. Support for populations of related genomes with variants (branching strings)

5. References
==

1. Basic BWT code in Matlab: http://schatzlab.cshl.edu/teaching/2012/BWT.m
2. Bowtie paper: http://genomebiology.com/2009/10/3/R25
3. FM Index: http://web.unipmn.it/~manzini/papers/focs00draft.pdf
4. BWT paper: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf