7.36/7.91 recitation

DG Lectures 5 & 6

2/26/14

Announcements

- project specific aims due in a little more than a week (March 7)
- Pset #2 due March 13, start early!
- Today:
  - library complexity
  - BWT and read mapping
  - de Bruijn graphs

Library Complexity

- You isolate DNA from your sample of interest (genomic DNA, ChIP-seq, cDNA from RNA-seq, etc.)
- You go through a protocol to generate a "**library**" of size-selected molecules (e.g. 200-300bp for paired-end genomic DNA sample, much smaller from ChIP-seq, etc.)
  - Library is only a subset of your original sample – some molecules have been lost due to:
    - 1. Stochastic sampling (molecules w/ low count are often lost)
    - 2. Systematic exclusion at steps of the library prep. protocol
- The number of *UNIQUE* molecules in your library is known as "library complexity", ***C*** (total # of molecules ***T*** is >> ***C*** due to PCR amplification during library prep.)
- The number of sequencing reads you get back is ***N*** (<*T* since only a subset of reads in the library cluster on the flow cell and are sequenced)

Library Complexity: Naïve approach

- Assume each unique molecule is represented equally in the library
  - Not a terrible assumption for genomic DNA (but still not true due to PCR amplification bias)
  - Definitely not a true for ChIP-seq, RNA-seq, etc. in which most frequent molecules in the sample (millions of copies) will have certain fragments represented multiple times in the library even after random fragmentation
  - Each molecule has prob. 1/***C*** of being sequenced. You sequence ***N*** total molecules, with ***M*** UNIQUE molecules in your sequencing run (***M*** < ***N***)
    - # times each molecule in the library sequenced follows a Poisson distribution with mean (= variance):

$$\lambda = \frac{N}{C}$$

    - P(observing a molecule in sequencing output) =1 – P(sequencing the read 0 times) =

$$\sum_{x=1}^{\infty} \frac{e^{-\lambda}\lambda^{x}}{x!} = 1 - \frac{e^{-\lambda}\lambda^{0}}{0!} = 1 - e^{-\lambda}$$

    - Maximum likelihood estimate of *C* is $\hat{C}$:

$$M = C \times P(\text{observing a molecule}) \implies \hat{C} = \frac{M}{1 - e^{-\lambda}}$$

    - Note that our formula for estimating *C* involves λ, but λ=N/C (it's a function of what we're trying to estimate!). So this Poisson formula is not as simple as it appears…would likely want to perform iterative estimation of these parameters.

Library Complexity: Naïve approach

- Testing this against real data (e.g. subsampling) reveals the assumption (each unique molecule is represented equally in the library) is totally off!
  - Again, due to: 1. Stochastic sampling of sample molecules into library
  - 2. Original highly expressed molecules are represented multiple times
  - 3. PCR amplification bias of some molecules
- Thus, λ is not the same for every molecule, and we should allow it to vary

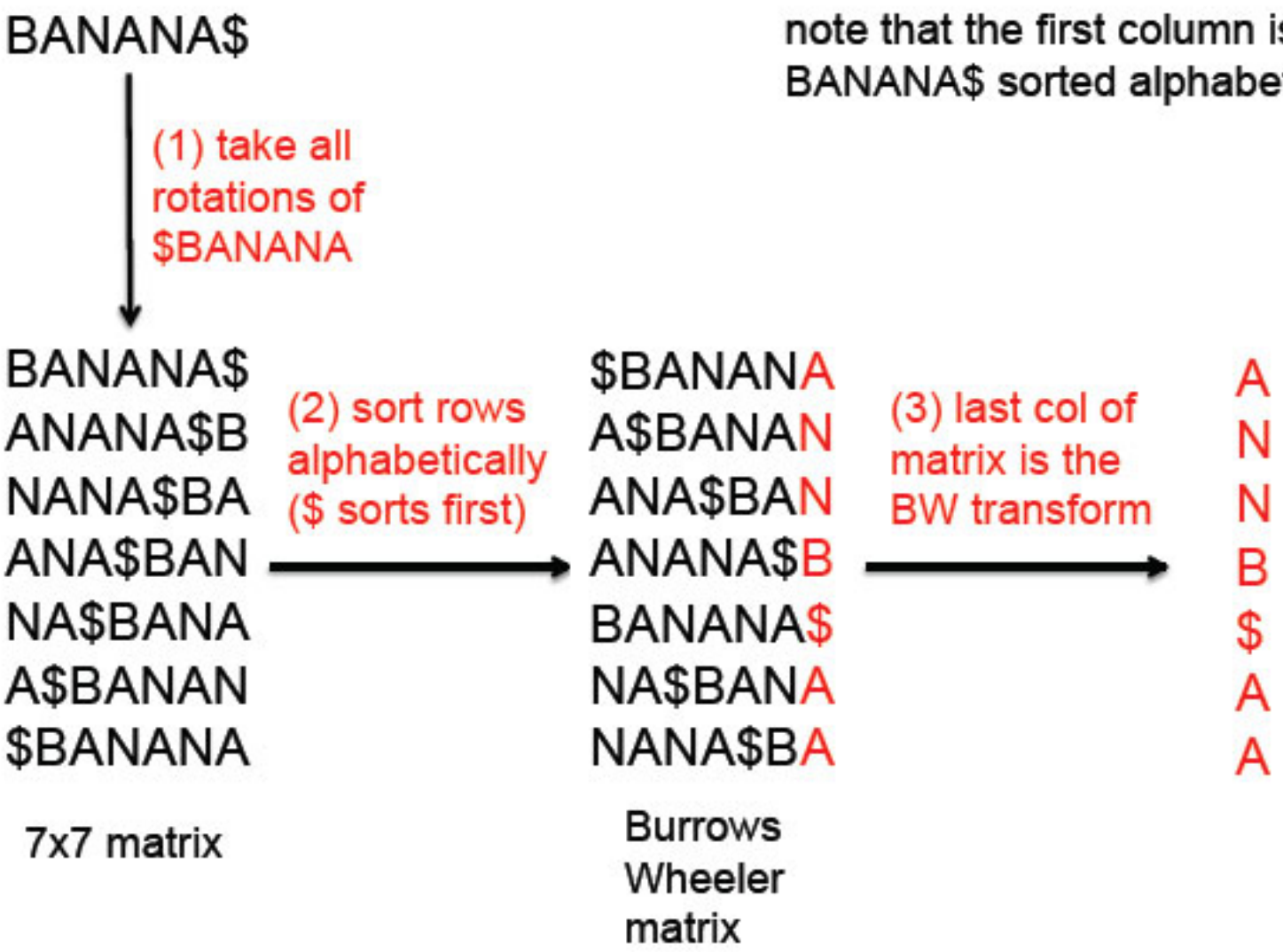
  - This motivates using the Negative Binomial (2 parameters) instead of the Poisson (1 parameter)
- Negative binomial distribution models an overdispersed Poisson distribution: useful when the variance > mean (they're equal in Poisson).
  - The dispersion is *k* (this needs to be fit to data)
  - Higher *k* ➔ more over-dispersed library ➔ more "biased" library and further deviation from assumption that each unique molecule is represented equally
  - Note that there are different parameterizations of the Negative Binomial: In terms of (λ and *k*) or (*r* and *p*) in lecture slides, as well as whether *r* is the desired success or the last failure. See http://www.johndcook.com/negative_binomial.pdf

  - Use Wikipedia/Matlab/WolframAlpha for Problem Set

Short read alignment (mapping)

- Motivation: Common sequencing experiments: ~100 million 100bp reads to align to a billion base pair genome
- Naïve ("ctrl-F" search) method of taking a read and searching the entire genome:
  - O(genome size = 1 billion) per read (without indels)
  - For all reads: O(1 billion x 200 million) – infeasible!
  - Ideally, something that approaches O(# of reads) – approximately independent of the size of genome
- We do this through BWT transform and FM index of genome

Burrows-Wheeler Transform (BWT)



Why use the BWT?

final char (L)	sorted rotations
a	n to decompress. It achieves compression
o	n to perform only comparisons to a depth
o	n transformation} This section describes
o	n transformation} We use the example and
o	n treats the right-hand side as the most
a	n tree for each 16 kbyte input block, enc
a	n tree in the output stream, then encodes
i	n turn, set $L[i]$ to be the
i	n turn, set $R[i]$ to the
o	n unusual data. Like the algorithm of Man
a	n use a single set of probabilities table
e	n using the positions of the suffixes in
i	n value at a given point in the vector $R
e	n we present modifications that improve t
e	n when the block size is quite large. Ho
i	n which codes that have not been seen in
i	n with ch appear in the {\em same order
i	n with ch. In our exam
o	n with Huffman or arithmetic coding. Bri
o	n with figures given by Bell~\cite{bell}.

Figure 1: Example of sorted rotations. Twenty consecutive rotations from the sorted list of rotations of a version of this paper are shown, together with the final character of each rotation.

we have grouped together rotations of our "genome" by their suffixes, and stored information about this ordering (the BWT)

e.g. all the rotations beginning with "n with" are grouped together

Burrows M, Wheeler DJ: A block sorting lossless data compression algorithm. *Digital Equipment Corporation, Palo Alto, CA* 1994, Technical Report 124; 1994

BWT and character rank

notice that each column and each row of the BW-matrix contains every letter from the input string

rank?			rank?
1	$BANANA	A	1
1	A$BANAN	N	1
2	ANA$BAN	N	2
3	ANANA$B	B	1
1	BANANA$	$	1
1	NA$BANA	A	2
2	NANA$BA	A	3

Burrows Wheeler matrix

Burrows Wheeler transform

- what do we mean by a character's rank in a column?

- the rank of a specific character *qc* is the number of *qc*s above it in the column, +1

- note that (for example) the "A" of rank 2 in the last column (BWT) and first column are the same lexical occurrence (the A preceded by "BAN" and followed by "NA")

- this lets us distinguish between the different "A" characters that occur in different *contexts* in the original string:

BANANA

rank in BWT? 3 2 1

- so, chars in the **F**irst and **L**ast columns have the same rank

Last to First (LF) function

LF(*i*, *qc*) = occ(*qc*) + count(*i*, *qc*)

returns the index of the char in the First column that is lexically equivalent to the instance of *qc* in position *i* of the BWT, using the fact that rank is same in First and Last columns

idx		BWT
0	$ B A N A N	A
1	A $ B A N A	N
2	A N A $ B A	N
3	A N A N A $	B
4	B A N A N A	$
5	N A $ B A N	A
6	N A N A $ B	A

occ(*qc*) = # characters lexically smaller than *qc* in the BWT (e.g. rank in **First** column of matrix)

count(*i*,*qc*) = # of *qc* characters before position *i* in the BWT (e.g. rank of *qc* character at position *i*, minus 1)

What is the output of LF(2,'N')? 6
occ('N')? 5
count(2,'N')? 1

Note: the character at position *i* in the BWT does not need to be *qc*! For example, what is:

LF(0,'N')? 5 (=5+0)

Last to First (LF) function

LF(*i*, *qc*) = occ(*qc*) + count(*i*, *qc*)

returns the index of the char in the First column that is lexically equivalent to the instance of *qc* in position *i* of the BWT, using the fact that rank is same in First and Last columns

idx	BWT
0	$ B A N A N A
1	A $ B A N A N
2	A N A $ B A N
3	A N A N A $ B
4	B A N A N A $
5	N A $ B A N A
6	N A N A $ B A

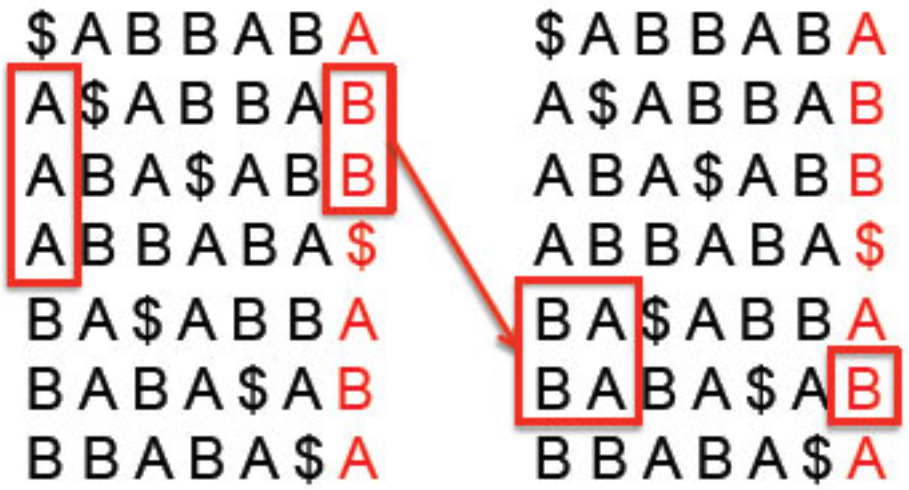
occ(*qc*) = # characters lexically smaller than *qc* in the BWT (e.g. rank in **First** column of matrix)

count(*i*,*qc*) = # of *qc* characters before position *i* in the BWT (e.g. rank of *qc* character at position *i*, minus 1)

In other words, since the first column is alphabetical, we first find the row where our character starts in the First column (e.g. Ns begin at index 5), then use the fact that rank is same in first and last column to find the specific char in First col that is the same lexical occurrences as our query in the BWT

Using the BWT and LF function to do exact matching

- say we have the following sequence: $ABBABA, and we want to find occurrences of BBA
- using the BW-matrix, we can start by finding all rows beginning with "A"



but, we're only interested in rows where the A is preceded by a B

and now, we only want suffixes where the B is preceded by another B

Once we know where our matches to our query begin in the BWT, use LF to get these locations in original string

To make this clearer, we showed the entire BW matrix. But how to do this if we only have the BWT?
- use the LF function!

http://www.cs.jhu.edu/~langmea/resources/lecture_notes/bwt_and_fm_index.pdf

Using the BWT and LF function to do exact matching

find occurrences of ANA in BANANA

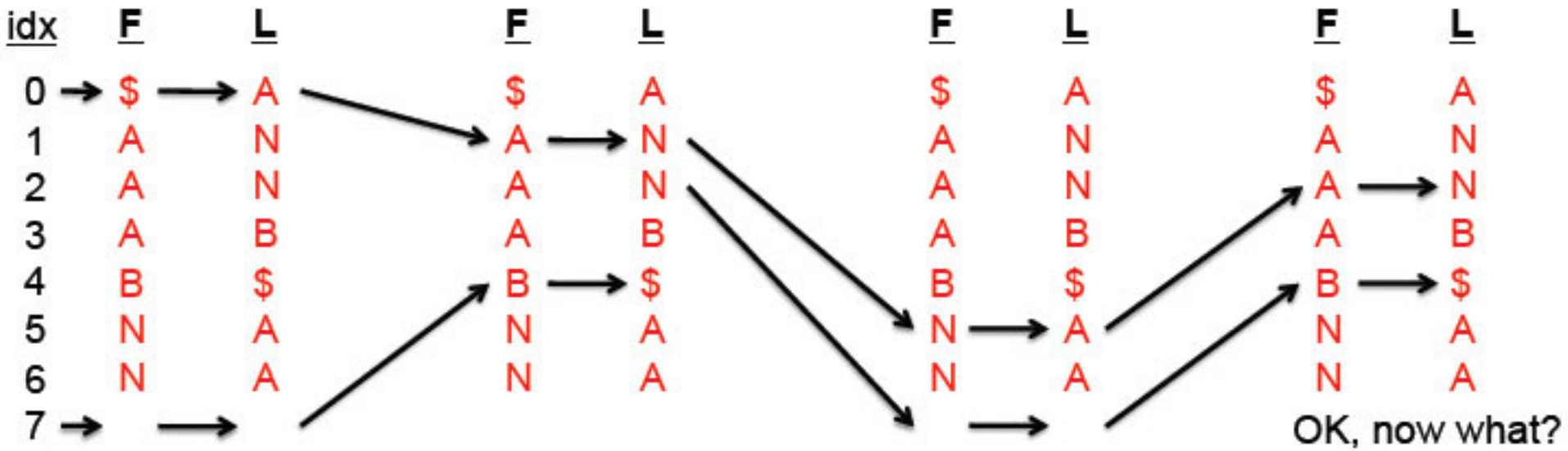
```
query = "ANA", bwt = "ANNB$AA"
top = 0
bot = len(bwt)                  # len(bwt) = 7 (yes, this is valid)
for qc in reverse(query):       # need to match from end to start of query
    top = LF(top, qc)           # LF(i,qc) = occ(qc) + count(i,qc)
    bot = LF(bot, qc)
```

iter 0:
top = 0
bot = 7

iter 1:
top=LF(0,'A')= **1**
bot=LF(7,'A')= **4**

iter 2:
top=LF(1,'N')= **5**
bot=LF(4,'N')= **7**

iter 3:
top=LF(5,'A')= **2**
bot=LF(7,'A')= **4**



Using the BWT and LF function to do exact matching

1) at end of loop, calculate **range** = bot – top
   - if **range** == 1:
     exactly one match at **top**
   - if **range** = *n* > 1:
     n matches of query at **top** to **bot**-1
   - if **range** == 0:
     query does not exist in genome

in our case, there are exactly two occurrences of "ANA" in "BANANA"

2) how can we find the location of our matches in the original string?
   - use LF function to walk back to the beginning of the string, starting at the beginning of the identified match (= **top**)
   - count # of times we "walk left" until hitting the end of string char ($) to get the hit offset

Find location of our match

- let's find the location of one of the two matches we found, corresponding to location **2** in the BWT:
- use LF to "walk back" from the beginning the match ("top") until we see the end-of-string character "$"

pseudocode:

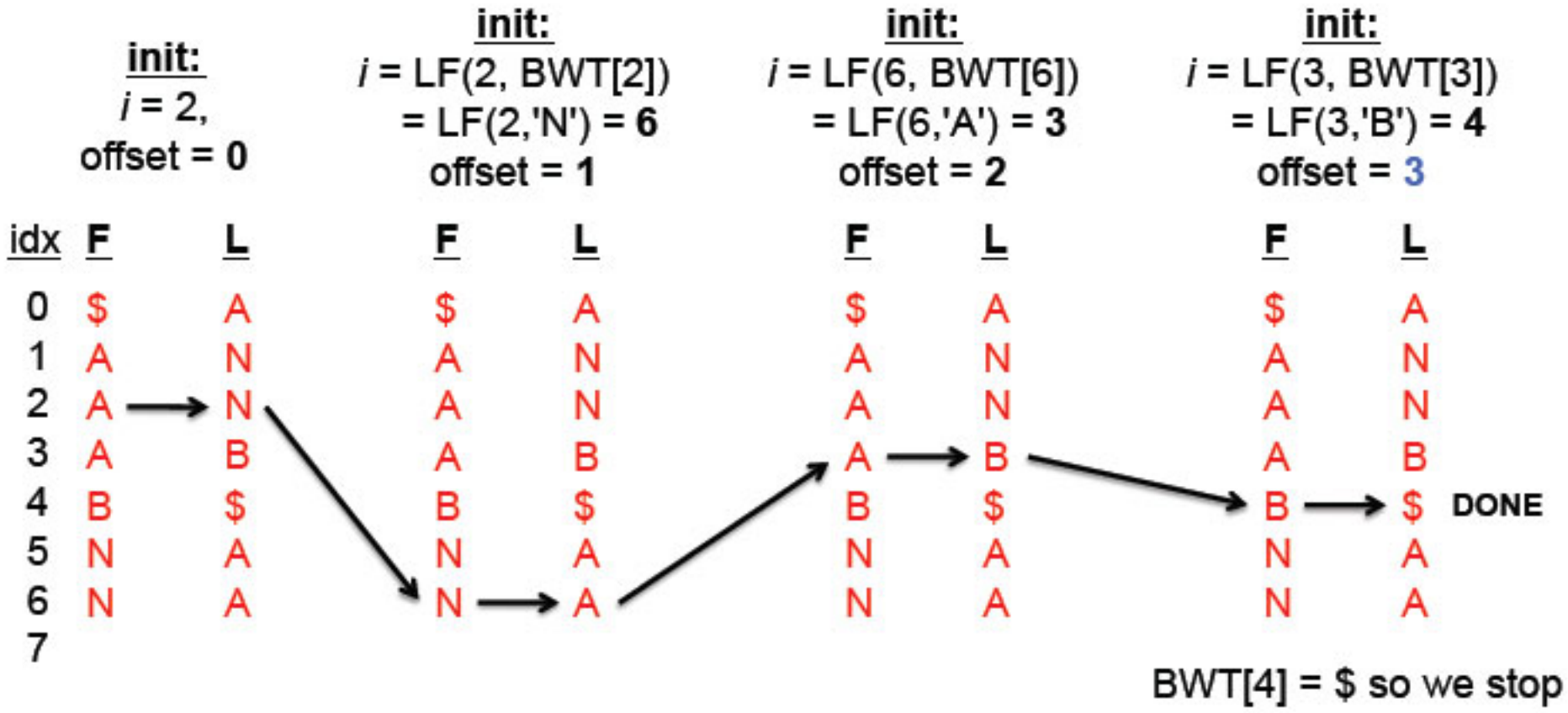
```
i = top                          # final value of 'top' after matching
offset = 0
while BWT[i] != "$":
    offset += 1
    i = LF(i, BWT[i])
```

Find location of our match

let's find the location of one of the two matches we found, corresponding to location *i*=***2*** in the BWT:

- use LF to "walk back" until we see the end-of-string char "$"
- obtain hit offset = 3 B A N A N A

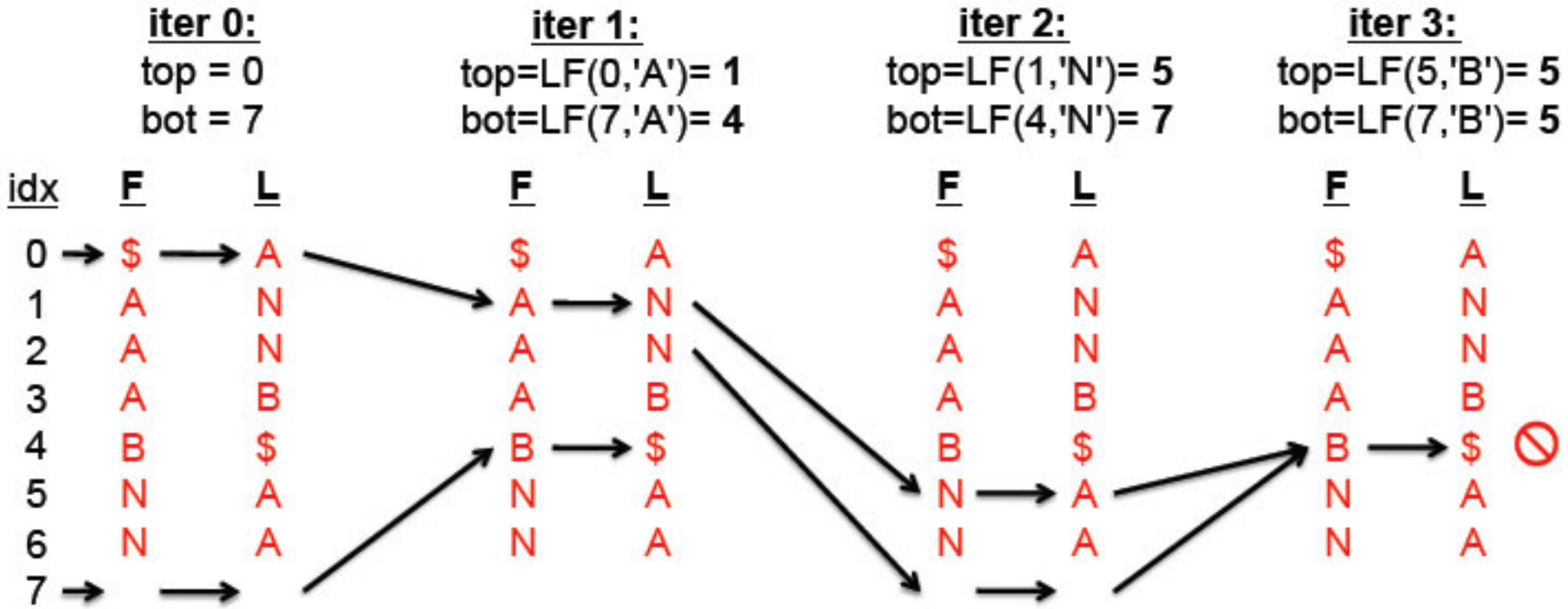
offset = 3



Using the BWT and LF function to do exact matching

if query does not exist in "genome", then top == bottom:

- find occurrences of BNA in BANANA:
- first 2 iterations are the same (*qc* = "A" and "N" respectively)
- at last iteration, we now have *qc* = "B"
- for both top and bot, occ = 4 and count = 1, top == bot == 5



Regenerating your string from its BWT

same as using LF to walk-back to the beginning of the string and get the hit offset, except you start from the end

- by definition, BWT[0] is the last character of original string

```
i = 0
str = ""
while bwt[i] != "$":
    str = bwt[i] + str
    i = LF(i, bwt[i])
```

0	A
1	N
2	N
3	B
4	$
5	A
6	A
7	

iter 0:
i = 0
str = ""

iter 1:
str = BWT[0] + str
= "A"
i = LF(0, "A") = **1**

iter 2:
str = BWT[1] + str
= "NA"
i = LF(1, "N") = **5**

iter 3:
str = BWT[5] + str
= "ANA"
i = LF(5, "A") = **2**

iter 4:
str = BWT[2] + str
= "NANA"
i = LF(2, "N") = **6**

iter 5:
str = BWT[6] + str
= "ANANA"
i = LF(6, "A") = **3**

iter 6:
str = BWT[3] + str
= "BANANA"
i = LF(3, "B") = **4**

BWT[**4**]
== "$"
DONE

Some caveats

- how to calculate occ(*qc*) and count(*i*,*qc*) when the genome is huge?
  - for occ(*qc*), need only to know where each character in the genome begins, for example as a dictionary in python:

    occ = {"A": 1, "C": 345, "G": 768, "T": 981}

    means there is only 1 char ("$") lexigraphically smaller than "A", there are 345 smaller than "C", 768 smaller than "G" etc.
  - count(*i*,*qc*) is trickier – naïvely, you could store for each ACGT a genome-length array containing count(*i*, *qc*) at location *i*, but these would be huge! Instead, store counts for only a subset of positions, then count # of *qc*s between *i* and closest stored count

- similarly, the method we showed for recovering the offset of a match using the LF function requires "walking back" to the beginning of the string – quite a long time, using a whole genome!
  - naïvely again, could keep genome-length mapping of BWT to original string indices -> lookup table
  - since genome is huge, instead store indices for every *i*th row

Together, these improvements comprise the FM-index

Genome Assembly ("shotgun sequencing")

- First sequencing of human genome
  - *De novo* assembly of sample from field work
- This is in contrast to most experiments in labs these days in which you generally are mapping your sequenced reads to the known reference genome of yeast, *C. elegans*, mouse, human, etc.
- May be situations in which you don't do full genome assembly but partial assembly for reads that don't map to reference genome to discover, for example, translocations, inversions, etc. in cancer tumor samples
- Longer reads and more depth is better, but limited by:
  - Experimental cost
  - Errors in longer reads (higher errors from bases ~50 onward)

Two main approaches

- Overlap layout consensus (string graph)
  - Plus: Retain entire read and its long-range position implications
  - Drawback: computationally expensive / slow
- de Bruijn graph
  - Plus: Computationally more tractable
  - Minus:
    - Graphs get messy (bubble, tips) and must use heuristics to trace path through graph
    - Lose longer-range position information: have 100bp reads and *k*=30

-Lose the fact that these two 30mers are separated by 40bp

Overlap graph

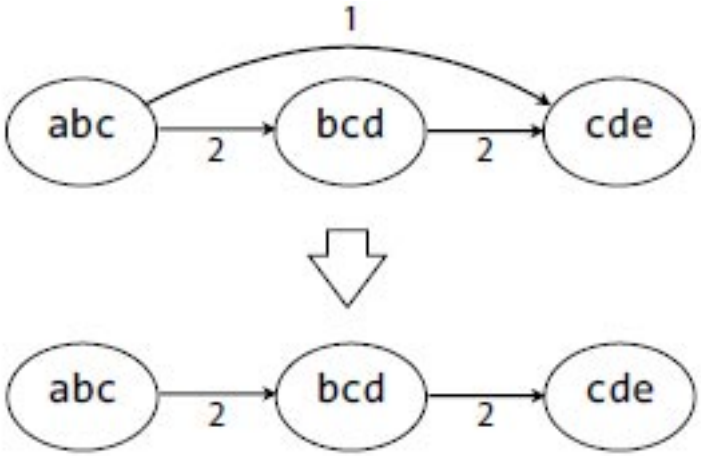
- Each node is a read
  - Directed edge is overlap between suffix of that read to prefix of another read (overlap at least length l)
  - Also include reverse complements of reads since we only sequence 1 of 2 strands of DNA

http://www.langmead-lab.org/teaching-materials/

  - To assemble the genome, we would ideally find the *shortest common superstring*
    - This is the *shortest* string that contains all relationships implied by the overlap graph from all of the reads
    - Finding a "Hamiltonian path" (traveling salesman problem) to visit all of the nodes: turns out this is too complex of a problem to solve
    - Greedy methods that will find not the shortest but close to shortest (bounded by ~2.5x shortest)
      - Greedy method makes the locally optimal choice at each step but doesn't look any further ahead to take steps that may lead to overall better solutions in the end
  - Even if it were tractable, one problem with this approach is that finding the *shortest common superstring* will collapse repeats longer than your overlap

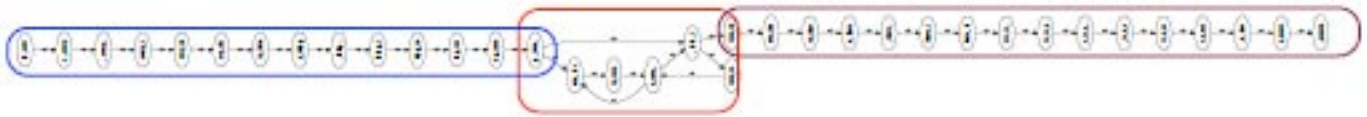
Overlap graph

- To simplify the graph, can remove edges that provide no additional information



http://www.langmead-lab.org/teaching-materials/

- This produces *contigs* of consecutive sequence: each contig corresponds to a clear path through part of the graph



Contig 1
to_every_thing_turn_

Contig 2
turn_there_is_a_season

Unresolvable repeat

From overlap graphs ➔ de Bruijn graphs

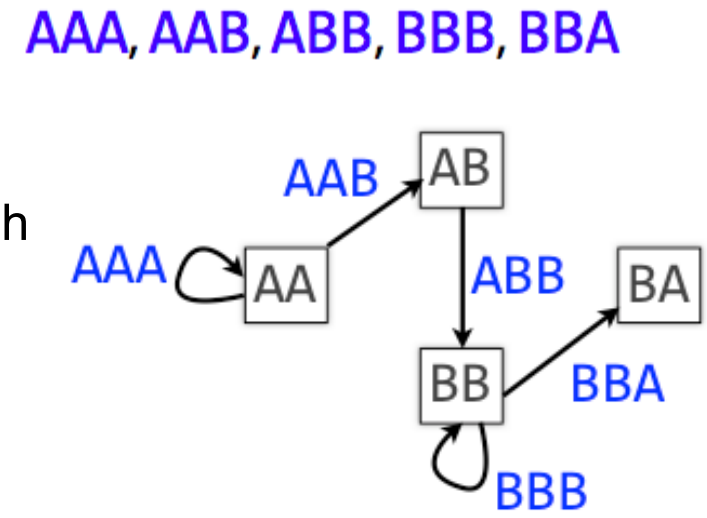
- Overlap graphs: each *k*mer is a node
  - Would like to visit each *node* once to assemble a version of the genome
    - Hamiltonian path through the graph
  - However, this very hard (NP-hard) and computationally does not scale to large genomes
- de Bruijn graphs: each *k*mer is an edge
  - In contrast, the problem of visiting each *edge* once to assemble a version of the genome is computationally tractable
    - Eulerian path through the graph

de Bruijn graph

- Choose *k* smaller than read length *L*
  - Tradeoff:
    - Smaller *k* loses more long-range information for repetitive regions and could create spurious overlaps if too small.
    - But too large *k* could eliminate edges between truly adjacent regions of genome if there are sequencing errors and/or low coverage of region.
    - Empirically, *k* is usually in 60s
    - *k* is odd so that no read is its reverse complement (middle base cannot be same in read and reverse complement)

de Bruijn graph

- Choose *k* smaller than read length *L*
  - Each read has *L*-*k*+1 *k*mers
    - For each of these *k*mers, consider its two (*k-1*)mers (prefix and suffix)
    - These (*k-1*)mers are the nodes of the graph
    - Connect the prefix (*k*-1)mer node to the suffix (*k*-1)mer node with a directed arrow



    - The edges are *k*mers from the input reads
      - If a *k*mer is present in multiple times in your reads, can draw one arrow weighted with the number of occurrences of that *k*mer in all of the reads

de Bruijn graph

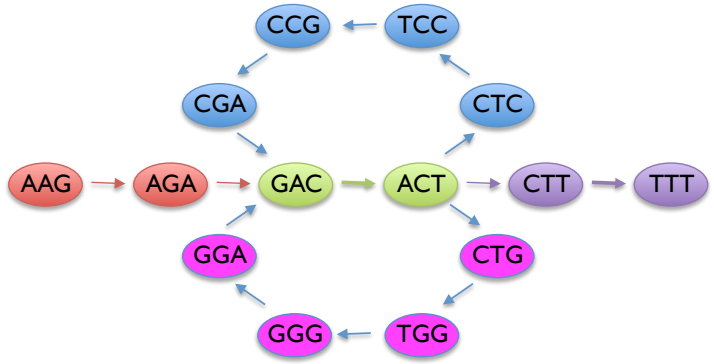
- Although computationally tractible, still are practical problems
  - Graph can have multiple walks through the graph; only one corresponds to true path
  - Gaps in coverage lead to *disconnected* graph (can only assemble two contigs)
  - Differences in coverage can lead to different # of incoming and outgoing arrows
  - Errors in reads

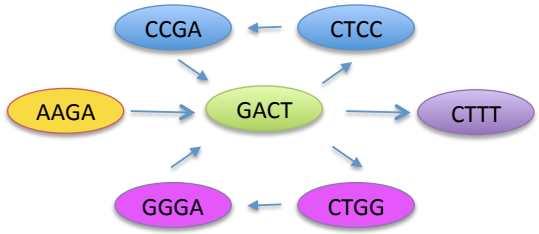
de Bruijn graph post-processing

- Before traversing the graph:
  - Simplify chains
  - Trim off ‘dead-end’ tips
  - Pop bubbles
  - Clip short, low-coverage nodes

de Bruijn graph post-processing

- Before traversing the graph:
  - Simplify chains
    - Collapse two nodes (or sets of nodes), one with only 1 outgoing arrow and the other with only 1 incoming arrow, into one longer n





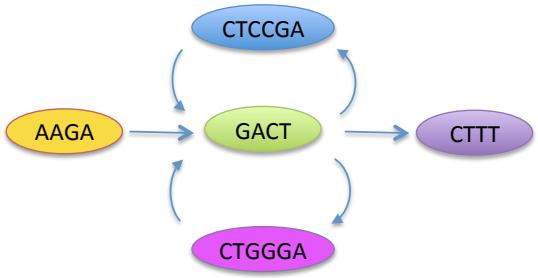


Figure adapted from presentation by Michael Schatz

de Bruijn graph post-processing

- Before traversing the graph:
  - Trim off ‘dead-end’ tips
    - Tip: short chain of nodes that is disconnected on one end
    - Caused by sequencing errors in read

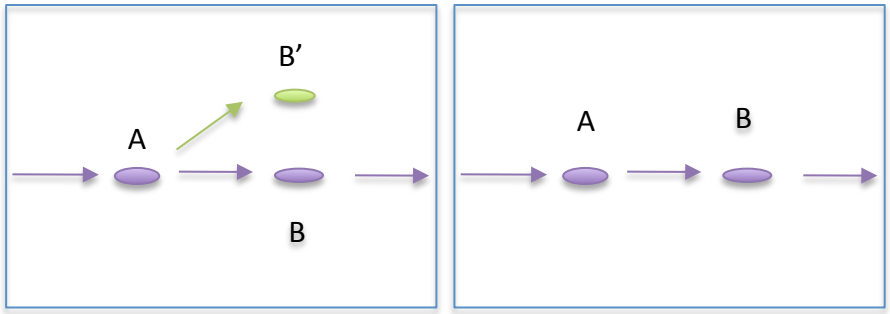


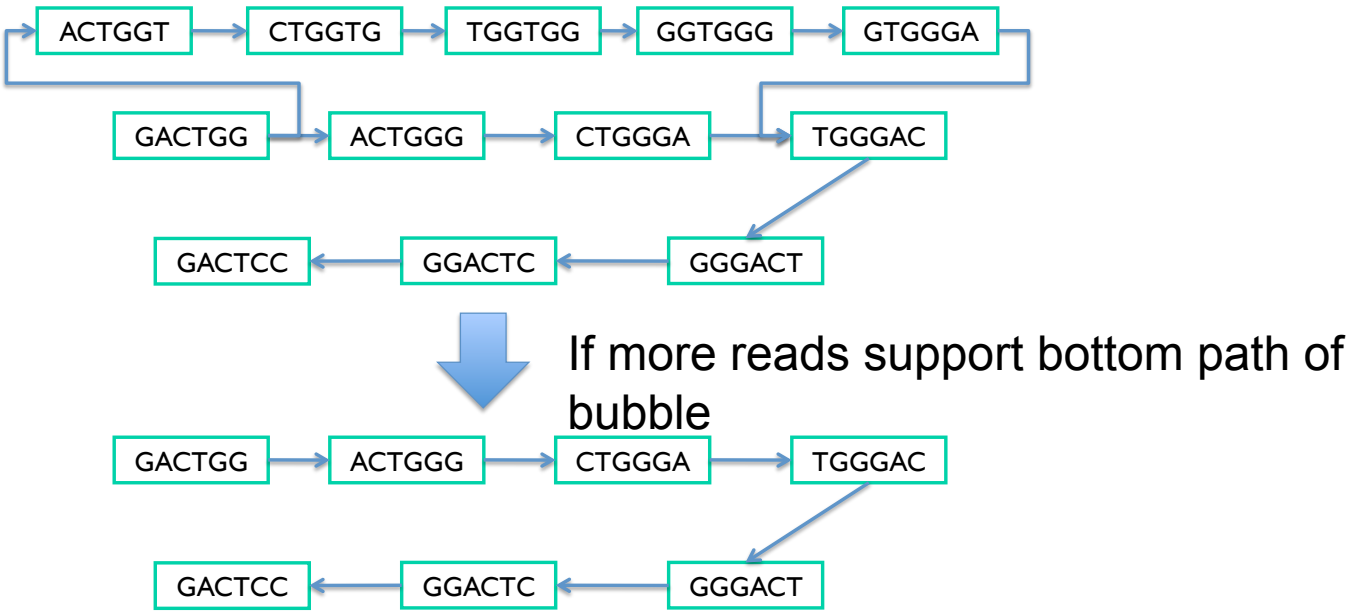
Figure adapted from presentation by Michael Schatz

de Bruijn graph post-processing

- Before traversing the graph:
  - Pop bubbles
    - Bubble: two paths that are redundant by starting and ending at the same nodes and contain similar sequences
    - Caused by sequencing error or biological variation
    - More complicated heuristics of how to pop them; for our purposes, manually inspect for sequencing error and pop

de Bruijn graph post-processing

- Before traversing the graph:
  - Pop bubbles



de Bruijn graph post-processing

- Before traversing the graph:
  - Clip short, low-coverage nodes
    - Low coverage means they're probably erroneous

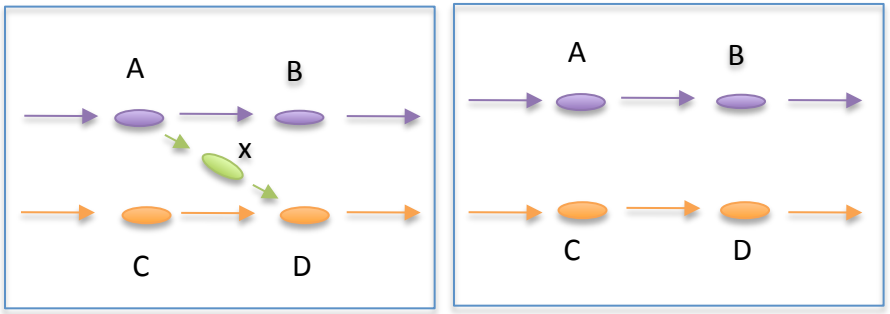


Figure adapted from presentation by Michael Schatz

de Bruijn graph

- Although some information is lost in de Bruijn graphs, they are fast and simple
  - Most commonly-used assemblers are de Bruijn graph-based

Nature Biotech primer on de Bruijn graphs:
http://www.cs.ucdavis.edu/~gusfield/cs225w12/deBruijn.pdf

7.91J / 20.490J / 20.390J / 7.36J / 6.802 / 6.874 / HST.506 Foundations of Computational and Systems Biology
Spring 2014