CS 262 Lecture 4: Burrows-Wheeler Transform

Winter 2015
Professor: Serafim Batzoglou
Notes scribed by Sanjay Siddhanti
*This lecture was given by Victoria Popic

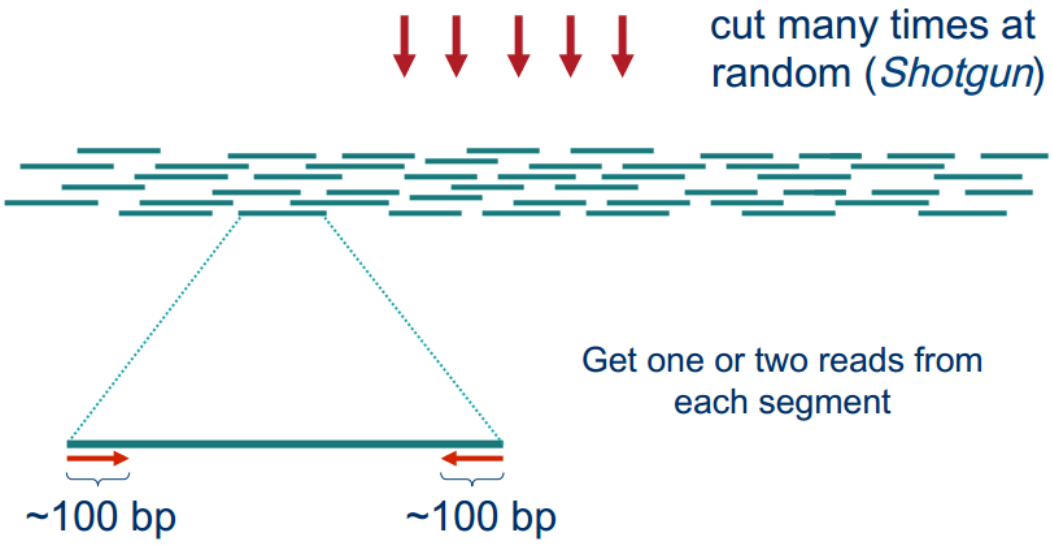
Contents

Topic 1: DNA Sequencing

- Goal: obtain the full nucleotide sequence of a piece of DNA
- This is challenging because no machine is currently capable of sequencing a long piece of DNA without first breaking it up
- Currently we can only sequence ~150 nucleotides at a time

Shotgun sequencing (current method):

- Input: a long piece of DNA
- Break the DNA strand at random locations to produce fragments of length ~100
    - Repeat this several times, breaking randomly each time, so that the result will be lots of overlapping fragments of length 100
    - Most of the human genome was sequenced to 12x coverage in the Human Genome Project (citation: http://en.wikipedia.org/wiki/Shotgun_sequencing)
- Sequence each of the short fragments and reassemble them using the overlap in the sequences
    - This is particularly difficult when there are short repeats of DNA



Assembly

- Genome assembly is just the process of attaining a genome sequence
- There are two main assembly problems:

1. **De Novo Assembly**

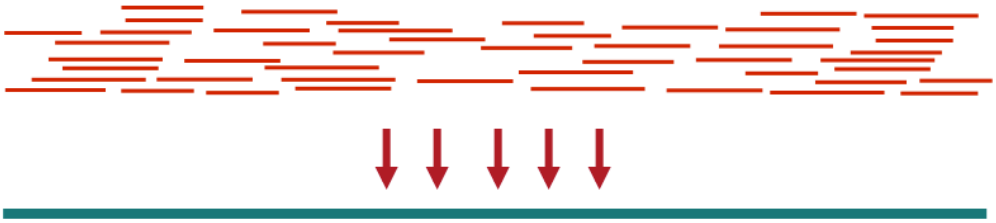
- First time sequencing an organism, so there is no template (reference genome) to compare against
- This problem is extremely challenging

2. **Resequencing**

- Sequencing an organism for which there exists a reference genome
- Every time a human genome is sequenced, this is an example of resequencing
- This problem is much easier because humans are genetically and biochemically very similar (http://en.wikipedia.org/wiki/Human_genetic_variation)
    - Instead of assembling only based on overlap, we can match short fragments to the reference genome to see where they belong – this is called **read mapping** (see below)

Read Mapping

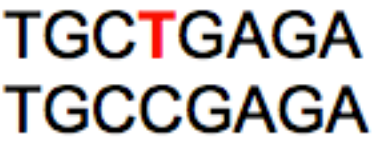
- Given a bunch of short fragments (reads) from shotgun sequencing, figure out where they belong on the template genome
- This is algorithmically challenging because naïve solutions will take linear time with respect to the reference genome
    - The human genome has 3 billion base pairs, so clearly using a linear time algorithm will be slow and not ideal
- This problem is further complicated by the fact that individuals differ, so sometimes a fragment might not be an exact match to anywhere in the reference genome
- We want something that is a) fast and b) detects genetic variation
- Modern fast read aligners include BWT, Bowtie, SOAP
    - All are based on the Burrows-Wheeler Transform



Topic 2: Human Genetic Variation

1. SNP = Single Nucleotide polymorphism

- A change at one nucleotide



2. Inversion

- A piece of DNA is reversed

3. Translocation

- Parts of non-homologous chromosomes rearrange

4. Microdeletion

- A small deletion (up to 5 MB)

TGC - - AGA
TGCCGAGA

5. Large deletion

- Deletion > 5 MB

6. Novel sequence

- Same as “Insertion” in the fragment

TGCTCGGAGA
TGC - - - GAGA

7. Mobile element insertion / Pseudogene insertion

- Mobile element = piece of DNA that can move around
- Pseudogene = nonfunctional relative of a functional gene

8. Tandem duplication

- Duplication of a piece of DNA in one strand

9. Transposition

- A piece of DNA gets moved to somewhere within the opposite strand

10. Novel sequence at breakpoint

- **DNA breakpoint** – locations in the genome where an inversion, deletion, etc is likely to occur (http://www.dnalc.org/view/1241-Breakpoints.html)

Topic 3: Burrows-Wheeler Transform

- Take a long sequence S, and a pattern P that you want to locate in S
- The BWT is a reversible permutation of S
    - Given the BWT, S can be reconstructed
- **The BWT allows one to search for pattern P in sequence S in O(|P|) time**
    - This means that the time required is linear with respect to the length of the PATTERN
    - Does NOT depend on the length of the sequence!

Constructing the BWT (with example word "BANANA")

<u>Method 1: Simple approach</u>

- There are better ways to get the BWT, but we will go through a simple approach
- Let $ be a character that is not in the alphabet, and is lexicographically smaller than all characters in the alphabet

1. Figure out all suffixes of the word

BANANA
ANANA
NANA
ANA
NA
A

2. Add '$' to the end of each suffix

BANANA$
ANANA$
NANA$
ANA$
NA$
A$
$

3. Add the prefix (where prefix + suffix = original word) after '$'

- Every line below is called a "rotation"

BANANA$
ANANA$B
NANA$BA
ANA$BAN
NA$BANA
A$BANAN
$BANANA

4. Sort the rotations lexicographically

$BANANA
A$BANAN
ANA$BAN
ANANA$B
BANANA$
NA$BANA
NANA$BA

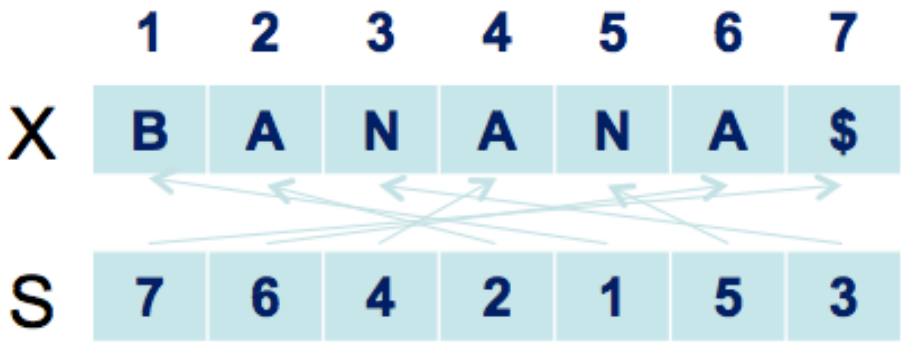
The BWT is the last column of the sorted matrix of rotations (highlighted in red)!

Method 2 for constructing BWT: Suffix Arrays

- The suffix array (http://en.wikipedia.org/wiki/Suffix_array) is a very useful data structure
- Sort all the suffixes of "BANANA$"

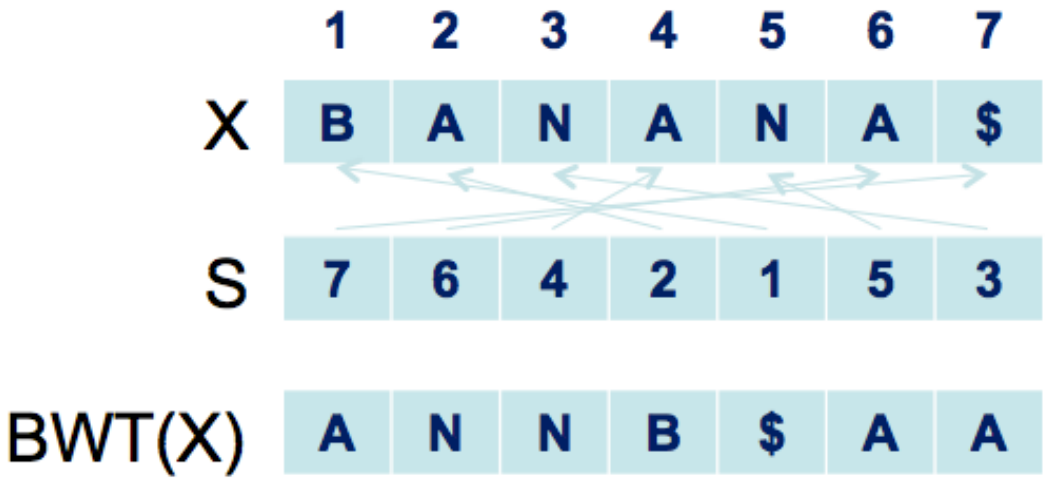
1 $BANANA
2 A$BANAN
3 ANA$BAN
4 ANANA$B
5 BANANA$
6 NA$BANA
7 NANA$BA

- Create a suffix array S for "BANANA$"
    - S[i] contains the index of where the ith smallest suffix occurs in the original string "BANANA$"
    - So S[1] will contain the index of where "$" (the smallest suffix, as seen above) occurs in "BANANA$". This is the last index, 7.
    - **Note: The lecture slides start the indexing at 1 (not 0), so I will continue this convention. S[1] is the first element of S.**
- Thus we end up with the following suffix array (bottom) for the original string (top)



Constructing BWT from Suffix Array:

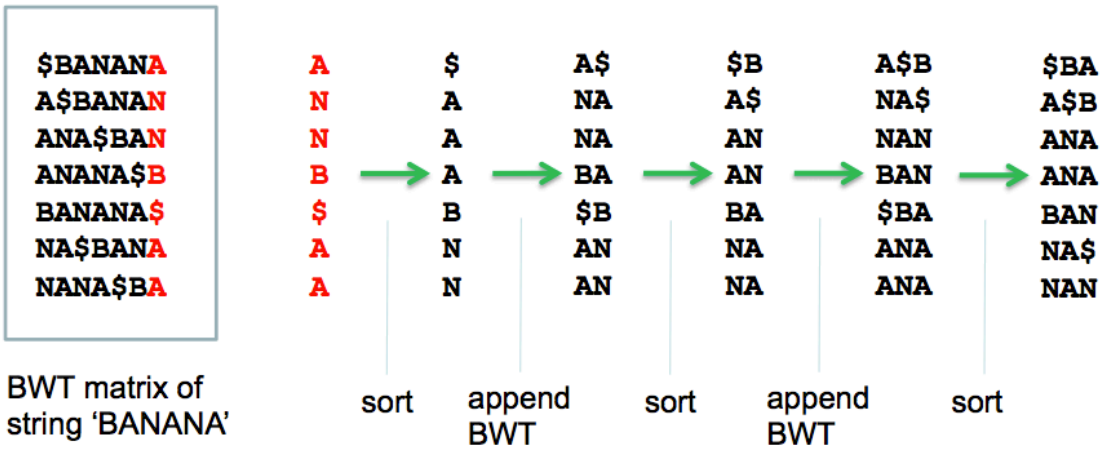
- Very simple rule to get BWT(X)[i], the ith character in the BWT of string X
    - In words: Go to the character in X pointed to by S[i], and move one spot to the left. If S[i] = 1, take the last character of X (which is the $)
    - In Python-like code: BWT(X)[i] = X[S[i] - 1] if S[i] > 1 else X[len(X)]



Reconstructing the original string from the BWT:

- We said that the BWT is a reversible permutation, which means that given only the BWT, we must be able to recreate the original string

Method 1: Simple approach



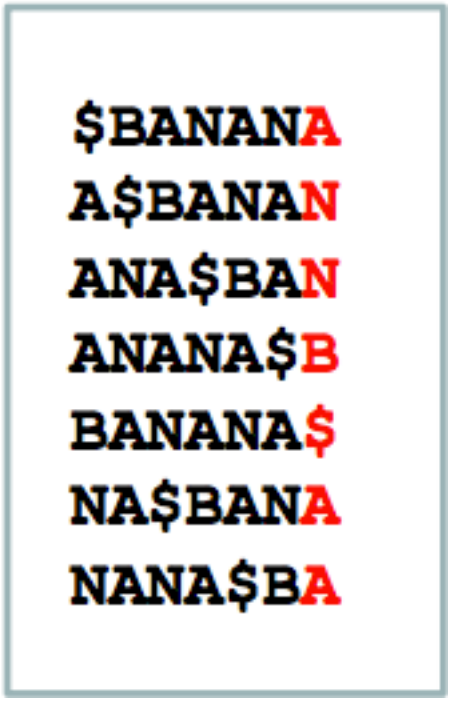
- Starting with an empty matrix, repeat the following procedure
    - Append BWT as last column of the matrix
    - Sort the matrix
- Do this n times, where n is the length of the BWT string
- After 1 iteration, this gives the first column of the sorted BWT matrix

$
A
A
A
B
N
N

- After 2 iterations, this gives the first two columns of the sorted BWT matrix

$B
A$
AN
AN
BA
NA
NA

- After n iterations, this gives the full BWT matrix

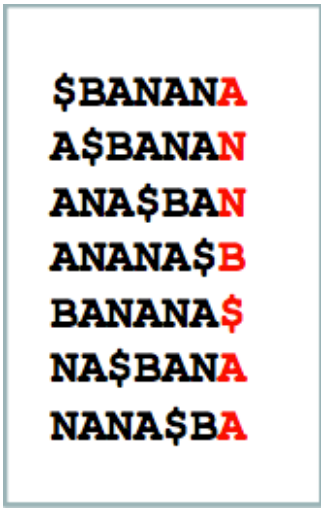


- The original string is just the first row, starting after the $

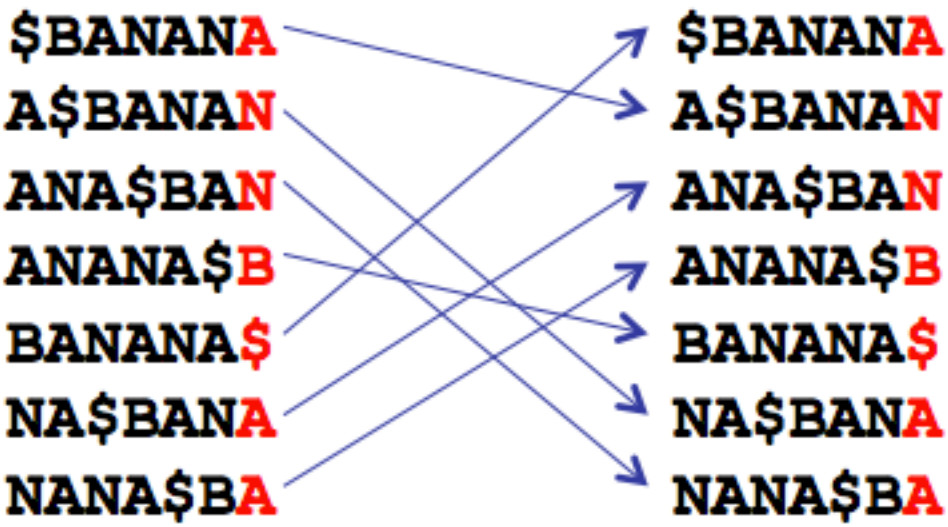
Method 2: Faster approach to reconstructing original string from BWT

Lemma:

- The i^{th} occurrence of character c in the last column of the BWT matrix is the same exact character as the i^{th} occurrence of character c in the first column
- Looking at our BWT matrix:



- The 1st occurrence of "A" in the last column (first row) is the third "A" in "BANAN**A**"
- The 1st occurrence of "A" in the first column (second row) is also this same exact "A", the third "A" in "BANAN**A**"
    - This will always be the case. It will never be a different "A".
- LF(r) is a function mapping from the last column of the BWT to the first column
    - Parameter r is the row number
    - In the last column of row r in the BWT matrix, say we have the ith occurrence of character j
    - LF(r) finds the ith occurrence of letter j in the first column of the BWT matrix, and returns that row



LF[] = [2, 6, 7, 5, 1, 3, 4]

- If LF(r) = x, then row x can be attained by rotating row r by one position to the right
    - This is obvious because of how LF is defined

- Computing LF is very simple
    - Let C('a') be the number of characters in the string smaller than 'a'
        - For banana:
            - C($) = 0
            - C(A) = 1 (only $ is smaller)
            - C(B) = 4 ($ and all three 'A's are smaller)
            - Etc..
    - LF(r) = C(X) + i, where row r holds the ith occurrence of letter X in the last column
- Now there is a very elegant way to reconstruct the original string
    - We notice that the first row of the BWT matrix is just the original string
        - So the first letter of the actual BWT is therefore the last letter of the original string
        - The idea is to work backwards from here – what is the 2nd to last character? If we compute LF(1st row) = row X, then we know that the very last character in row X = BWT[X] = 2nd to last character in original string.
        - Then compute LF(X) = Y, and BWT[Y] must be the 3rd to last character in the original string
        - Continue doing this until we hit a "$" in the BWT – this means we have reconstructed the entire string

	A	N	N	B	$	A	A	
C()	1	5	5	4	0	1	1	C() copied for convenience
index i	1	1	2	1	1	2	3	indicating this is i-th occurrence of 'c'
LF()	2	6	7	5	1	3	4	LF() = C() + i

Reconstruct BANANA:

```
S := ""; r := 1; c := BWT[r];
UNTIL c = '$' {
        S := cS;
        r := LF(r);
        c := BWT(r); }
```

- This reconstruction takes O(n) time

Searching for a pattern in the BWT:

- L(W): lowest index in BWT matrix where W is prefix
- U(W): highest index in BWT matrix where W is prefix
    - Ex: L("NA") = 6, U("NA") = 7 for BANANA

Lemma

- L(aW) = C(a) + i+1, where i = # of occurrences of character 'a' up to slot L(W)–1 in BWT (X)
- U(aW) = C(a) + j, where j = # of occurrences of character 'a' up to slot U(W) in BWT (X)

- L("ANA") = C('A') + # 'A's up to (L("NA") – 1) + 1
    - = 1 + (# 'A's up to 5) + 1
    - = 1 + 1 + 1 = 3
- U("ANA") = 1 + # 'A's up to U("NA") = 1 + 3 = 4
- Then we can find where our pattern occurs using the following algorithm:

```
Let
LFC(r, a) = C(a) + i, where i = #'a's up to r in BWT

ExactMatch(W[1…k]) {

    a := W[k];
    low := C(a) +1;
    high := C(a+1);    // a+1: lexicographically next char
    i := k – 1;
    while (low <= high && i >= 1) {
            a = W[i];
            low = LFC(low – 1, a) + 1;
            high = LFC(high, a);
            i := i – 1; }
    return (low, high);
}
```

- We can thus search for all exact occurrences of W in time O(|W|)

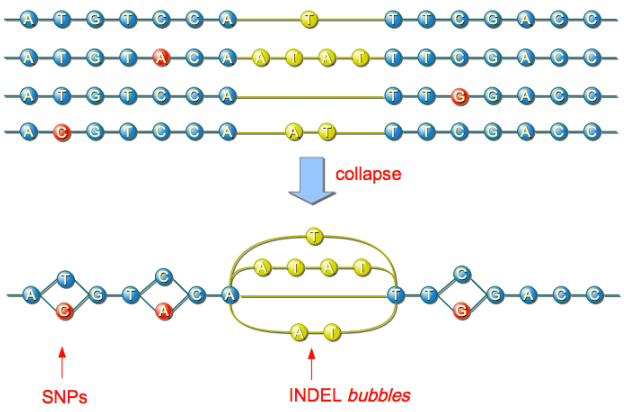
BWT-based aligners in practice

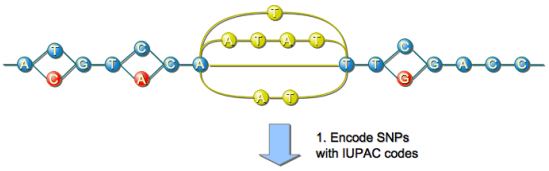
- Inexact matching
    - Allow mismatches and gaps
    - This accounts for sequencing errors, mutations, etc

- Heuristics
    - Put bounds on the number of allowed differences
    - Add heuristics for scoring (different types of gap penalties, different confusion matrix for mismatch penalties, etc.)
- Memory optimizations
    - Precompute parts of the suffix or occurrence arrays if necessary
- Aligners often have 3 phases:
    1. BWT index construction
    2. Short read mapping
    3. Reporting and evaluation of alignment results

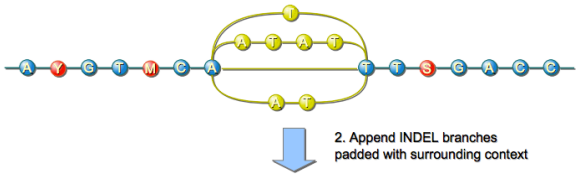
Topic 4: Short read alignments with populations of genomes

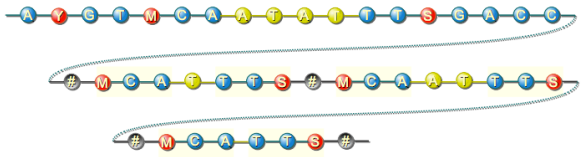
- For many species, there are now an abundance of genomes available
    - 1000 Genomes Project in humans
    - 1000 Plant Genomes Project
    - 100K Genomes Project for infectious microorganisms
    - i5k Project for arthropods
    - 1001 Genomes Project for A. Thaliana (a flowering plant)
    - etc…
- When we do short read alignments to just one reference genome, that comes with whatever biases are in the reference genome
    - To avoid this, we can even out the biases by aligning to populations of genomes
- **Idea**: create a compressed reference representation (reference multi-genome) that captures all the variations in the genome collection and can be used for short-read alignment
    - see pictures below





IUPAC	-	A	B	C	D	G	H	K	M	N	R	S	T	V	W	Y
base	#	A	C|G|T	C	A|G|T	G	A|C|T	G|T	A|C	A|C|G|T	A|G	C|G	T	A|C|G	A|T	C|T





- now we can use this in modified alignment algorithms