Read Mapping

Burrows Wheeler Transform and Reference Based Assembly

Peter N. Robinson

Institut für Medizinische Genetik und Humangenetik
Charité Universitätsmedizin Berlin

Genomics: Lecture #5 WS 2014/2015

Today

- Burrows Wheeler Transform
- FM index
- Burrows Wheeler Aligner (bwa)

Outline

Burrows Wheeler Transform (BWT)

The BWT applies a reversible transformation to a block of input text. The transformation does not itself compress the data, but reorders it to make it easy to compress with simple algorithms such as move-to-front coding.

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

- Basis for the bzip2 compression algorithm
- Basis for many of the read mapping algorithms in common use today

Burrows Wheeler Transform (BWT)

- The significance of the BWT for most of the rest of the world is as a data compression technique
- However, the BWT leads to a block-sorted data structure that is well suited to searching short strings in a larger text.
- The FM index uses the BWT to enable search with time linear in the length of the search string.

  Ferragina P, Manzini P (2000) Opportunistic Data Structures with Applications.

  Proceedings of the 41st IEEE Symposium on Foundations of Computer Science
- Today, we will explain the BWT and then the FM index and show how they are used in bwa for read alignment.

Burrows Wheeler Transform (BWT)

First step: form all **rotations** of the input text, which we will call T. Note that as with the suffix array and suffix tree, we append a **termination character** $ to the end of the text

```
T="abracadabra$“

 0: abracadabra$
 1: bracadabra$a
 2: racadabra$ab
 3: acadabra$abr
 4: cadabra$abra
 5: adabra$abrac
 6: dabra$abraca
 7: abra$abracad
 8: bra$abracada
 9: ra$abracadab
10: a$abracadabr
11: $abracadabra
```

Burrows Wheeler Transform (BWT)

Second step: Sort the rotated strings lexicographically

```
 0: abracadabra$              0: $abracadabra
 1: bracadabra$a              1: a$abracadabr
 2: racadabra$ab              2: abra$abracad
 3: acadabra$abr              3: abracadabra$
 4: cadabra$abra              4: acadabra$abr
 5: adabra$abrac     sort     5: adabra$abrac
 6: dabra$abraca   ------>    6: bra$abracada
 7: abra$abracad              7: bracadabra$a
 8: bra$abracada              8: cadabra$abra
 9: ra$abracadab              9: dabra$abraca
10: a$abracadabr             10: ra$abracadab
11: $abracadabra             11: racadabra$ab
```

recall that the termination character $ comes before every other character lexicographically.

Burrows Wheeler Transform (BWT)

Third step: The **Burrows Wheeler Transform** is simply the last column of the **Burrows Wheeler matrix**.

```
$abracadabra
a$abracadabr
abra$abracad
abracadabra$
acadabra$abr
adabra$abrac
bra$abracada
bracadabra$a
cadabra$abra
dabra$abraca
ra$abracadab
racadabra$ab
```

Burrows Wheeler Transform (BWT)

We will denote the Burrows Wheeler transform of an input string T as

$$\mathrm{BWT}(T)$$

- Thus, $\mathrm{BWT}(T)$="ard\$rcaaaabb"
- It is relatively easy to implement a naive version of the BWT
    1. Create all rotations of T
    2. Sort the rotations lexicographically
    3. Concatenate the last character of each rotation to form BWT(T)

Burrows Wheeler Transform (BWT)

The BWT tends to contain lots of “runs” of identical characters, which is a good feature to have for compression algorithms such as **run-length encoding**.

- This is slightly difficult to appreciate with the short strings we are using for the slides, but consider the following excerpt of BWT(*Macbeth*, Act 1, Scene 1):

```
...uoaoiiiiiiiiiiiiiiiaaaaaiiiiiuiiiiiiiiiiiiiiiiiaAAiiiiiiioieei...
```

A simple run-length encoding might be

```
...uoaoi{15}a{5}i{5}ui{17}aA{2}i{7}oie{2}i...
```

BWT and Suffix array

BW matrix	Suffix array with corresponding suffixes	
$abracadabra	[11]	$
a$abracadabr	[10]	a$
abra$abracad	[7]	abra$
abracadabra$	[0]	abracadabra$
acadabra$abr	[3]	acadabra$
adabra$abrac	[5]	adabra$
bra$abracada	[8]	bra$
bracadabra$a	[1]	bracadabra$
cadabra$abra	[4]	cadabra$
dabra$abraca	[6]	dabra$
ra$abracadab	[9]	ra$
racadabra$ab	[2]	racadabra$

- The Burrows Wheeler matrix is (nearly) the same as the suffixes referred to by the suffix array of the same string

BWT and Suffix array

We can now write an algorithm to create BWT(T) from the suffix array of T. SA(T), by noting that position *i* of the BWT corresponds to the character that is just to the left of the *i*th suffix in the original string.
This character is "rotated" around to the back of the BW matrix

BW matrix	**Suffix array with corresponding suffixes**

```
Consider the fourth sorted rotation in the BWM and
the fourth suffix in the suffix array for T=abracadabra$
```

abra$abracad	[7] abra$

The character just to the left of the suffix is the i^{th} character of BWT(*T*)

T=abraca**d**abra$

BWT and Suffix array

BW matrix **Suffix array with corresponding suffixes**

Consider the fourth sorted rotation in the BWM and the fourth suffix in the suffix array for *T*=abracadabra$

abra$abracad [7] abra$

The character just to the left of the suffix is the *i*th character of BWT(*T*)

T=abracadabra$

- We can now construct the BWT as follows

$$\mathrm{BWT}(T) = \begin{cases} T[SA[i]-1] & \text{if} \quad SA[i] > 0 \\ \$ & \text{if} \quad SA[i] = 0 \end{cases} \qquad (1)$$

To see the reason for the second case, consider that the first suffix of the suffix array is always $

BWT and Suffix array

```
T=abracadabra$
  012345678901
```

$$\mathrm{BWT}(T) = \begin{cases} T[SA[i]-1] & \text{if} \quad SA[i] > 0 \\ \$ & \text{if} \quad SA[i] = 0 \end{cases} \tag{2}$$

BW matrix	Suffix array with corresponding suffixes	
$abracadabra	[11]	$
a$abracadabr	[10]	a$
abra$abracad	[7]	abra$
abracadabra$	[0]	abracadabra$
acadabra$abr	[3]	acadabra$
adabra$abrac	[5]	adabra$
bra$abracada	[8]	bra$
bracadabra$a	[1]	bracadabra$
cadabra$abra	[4]	cadabra$
dabra$abraca	[6]	dabra$
ra$abracadab	[9]	ra$
racadabra$ab	[2]	racadabra$

(Work through example)

Constructing a BWT from a Suffix Array

The naive algorithm is pretty simple to implement

Algorithm 1 bwtFromSuffixArray(T)

```
1:  $sa = \text{constructSuffixArray}(T\$)$
2:  $L = \text{length}(sa)$
3:  $bwt = \text{new string}[L]$
4:  for i=0 to i=L-1 do
5:      if $sa[i] = 0$ then
6:          $bwt[i] = \$$
7:      else
8:          $bwt[i] = T[sa[i] - 1]$
9:      end if
10: end for
11: return bwt
```

Reversing the BWT

If we have used the BWT to compress a string, and now want to get the original string back, we need to

1. Reverse the compression procedure (e.g., run-length encoding)
2. Get the original string back from the BWT

So, how *do* we reverse the Burrows Wheeler transformation?

The reversibility of the BWT depends on the

LF Mapping property

For any character, the T-ranking of characters in the first column (**F**) is the same as order of characters in the last column (**L**)

Reversing the BWT

So, what is the T-ranking?

$$a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$$$

- The T-ranking of the character at any given position is the number of times that an identical character has preceeded it in T
- The T-ranking of $ is always zero and is omitted here
- The ranks shown just to help understand the LF mapping property, they are not stored explicitly

Reversing the BWT

$$\begin{array}{l}
\$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \\
a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 \\
a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 \\
a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 \\
a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 \\
a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 \\
b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 \\
b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 \\
c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 \\
d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 \\
r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 \\
r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0
\end{array}$$

- Here is the Burrows Wheeler matrix with the T-ranks of all the characters.

Reversing the BWT

$$\begin{array}{l}
\$_0a_0b_0r_0a_1c_0a_2d_0a_3b_1r_1a_4 \\
a_4\$_0a_0b_0r_0a_1c_0a_2d_0a_3b_1r_1 \\
a_3b_1r_1a_4\$_0a_0b_0r_0a_1c_0a_2d_0 \\
a_0b_0r_0a_1c_0a_2d_0a_3b_1r_1a_4\$_0 \\
a_1c_0a_2d_0a_3b_1r_1a_4\$_0a_0b_0r_0 \\
a_2d_0a_3b_1r_1a_4\$_0a_0b_0r_0a_1c_0 \\
b_1r_1a_4\$_0a_0b_0r_0a_1c_0a_2d_0a_3 \\
b_0r_0a_1c_0a_2d_0a_3b_1r_1a_4\$_0a_0 \\
c_0a_2d_0a_3b_1r_1a_4\$_0a_0b_0r_0a_1 \\
d_0a_3b_1r_1a_4\$_0a_0b_0r_0a_1c_0a_2 \\
r_1a_4\$_0a_0b_0r_0a_1c_0a_2d_0a_3b_1 \\
r_0a_1c_0a_2d_0a_3b_1r_1a_4\$_0a_0b_0
\end{array}$$

- What do you notice about the T-ranks of the a characters?

Reversing the BWT

- The a's have the same relative order in the F and the L columns
- A similar observation pertains to the other characters

$$\begin{array}{l}
\$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \\
a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 \\
a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 \\
a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 \\
a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 \\
a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 \\
b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 \\
b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 \\
c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 \\
d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 \\
r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 \\
r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0
\end{array}$$

Reversing the BWT

$$\begin{array}{c|cccccccccccc}
\$_0 & a_0 & b_0 & r_0 & a_1 & c_0 & a_2 & d_0 & a_3 & b_1 & r_1 & a_4 \\
a_4 & \$_0 & a_0 & b_0 & r_0 & a_1 & c_0 & a_2 & d_0 & a_3 & b_1 & r_1 \\
a_3 & b_1 & r_1 & a_4 & \$_0 & a_0 & b_0 & r_0 & a_1 & c_0 & a_2 & d_0 \\
a_0 & b_0 & r_0 & a_1 & c_0 & a_2 & d_0 & a_3 & b_1 & r_1 & a_4 & \$_0 \\
a_1 & c_0 & a_2 & d_0 & a_3 & b_1 & r_1 & a_4 & \$_0 & a_0 & b_0 & r_0 \\
a_2 & d_0 & a_3 & b_1 & r_1 & a_4 & \$_0 & a_0 & b_0 & r_0 & a_1 & c_0 \\
b_1 & r_1 & a_4 & \$_0 & a_0 & b_0 & r_0 & a_1 & c_0 & a_2 & d_0 & a_3 \\
b_0 & r_0 & a_1 & c_0 & a_2 & d_0 & a_3 & b_1 & r_1 & a_4 & \$_0 & a_0 \\
c_0 & a_2 & d_0 & a_3 & b_1 & r_1 & a_4 & \$_0 & a_0 & b_0 & r_0 & a_1 \\
d_0 & a_3 & b_1 & r_1 & a_4 & \$_0 & a_0 & b_0 & r_0 & a_1 & c_0 & a_2 \\
r_1 & a_4 & \$_0 & a_0 & b_0 & r_0 & a_1 & c_0 & a_2 & d_0 & a_3 & b_1 \\
r_0 & a_1 & c_0 & a_2 & d_0 & a_3 & b_1 & r_1 & a_4 & \$_0 & a_0 & b_0
\end{array}$$

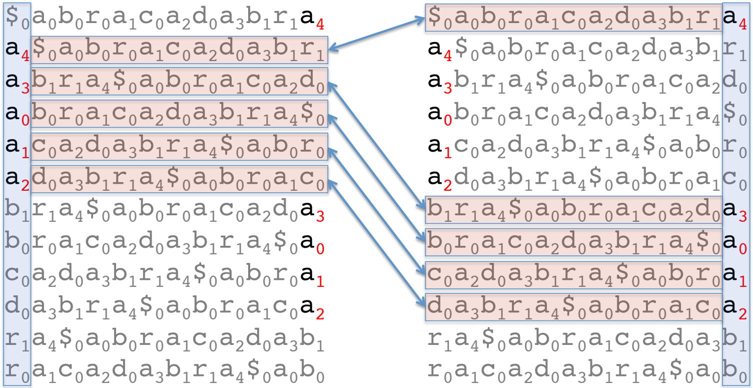
- The relative T-ranks of the a characters in column **F** are determined by the lexicographic ranks of the strings **to the right** of the characters

Reversing the BWT

$$\begin{array}{l|l}
\$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 & a_4 \\
a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 & r_1 \\
a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 & d_0 \\
a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 & \$_0 \\
a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0 & r_0 \\
a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 & c_0 \\
b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 & a_3 \\
b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 & a_0 \\
c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 & a_1 \\
d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 & a_2 \\
r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 & b_1 \\
r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 & b_0
\end{array}$$

- The relative T-ranks of the a characters in column **L** must reflect the lexicographic ranks of the strings **to the “rotated” right** of the characters

Reversing the BWT


$\$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4$
$a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1$
$a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0$
$a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0$
$a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0$
$a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0$
$b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3$
$b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0$
$c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1$
$d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2$
$r_1 a_4 \$_0 a_0 b_0 r_0 a_1 c_0 a_2 d_0 a_3 b_1$
$r_0 a_1 c_0 a_2 d_0 a_3 b_1 r_1 a_4 \$_0 a_0 b_0$

- These are the same strings (consequence of the rotation!)

Reversing the BWT

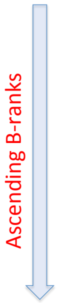
- We introduce another "vertical" ranking
- The B-ranking of a character at a specific position is the number of that times the same character has occured in the F column "above" the current position
- The B-ranking is thus like a cumulative count of the characters

$\$_0$abracadabra$_0$
a_0\$abracadabr$_0$
$a_1$bra\$abracad$_0$
$a_2$bracadabra\$$_0$
$a_3$cadabra\$abr$_1$
$a_4$dabra\$abrac$_0$
$b_0$ra\$abracada$_1$
$b_1$racadabra\$a$_2$
$c_0$adabra\$abra$_3$
$d_0$abra\$abraca$_4$
r_0a\$abracadab$_0$
$r_1$acadabra\$ab$_1$

Just the F and L columns are shown for better legibility

Reversing the BWT

- Column **F** has a simple structure: Chunks of identical characters with ascending B-ranks
- Column **L** does not generally have this kind of strict chunk structure, but the B-ranks of any given character also are arranged in ascending order

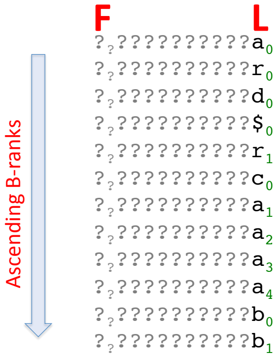


F **L**

$\$_0\texttt{abracadabr}a_0$
$a_0\texttt{\$abracadab}r_0$
$a_1\texttt{bra\$abraca}d_0$
$a_2\texttt{bracadabra}\$_0$
$a_3\texttt{cadabra\$ab}r_1$
$a_4\texttt{dabra\$abra}c_0$
$b_0\texttt{ra\$abracad}a_1$
$b_1\texttt{racadabra\$}a_2$
$c_0\texttt{adabra\$abr}a_3$
$d_0\texttt{abra\$abrac}a_4$
$r_0\texttt{a\$abracada}b_0$
$r_1\texttt{acadabra\$a}b_1$

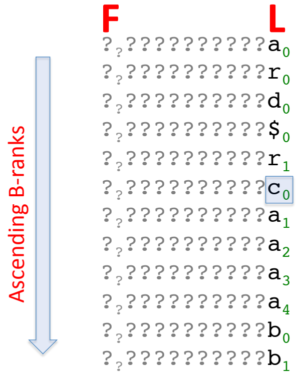
Reversing the BWT

- Can we now use these observations to reconstruct the original string?
- We will first try to reconstruct the first column of the BWM



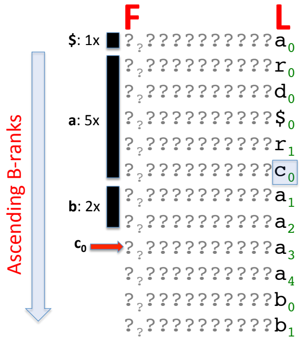
Reversing the BWT

- Consider c_0.
- We know that the \$, all the a's, all the b's, but not any of the d's must precede c_0 in the first column



Reversing the BWT

- The index of c_0 in column **F** must equal $1+5+2=8$
- We will refer to this as the **cumulative index property**



Reversing the BWT

- We will reconstruct the string from **right to left**
- We know the last character is $, so we initialize our reconstructed string accordingly

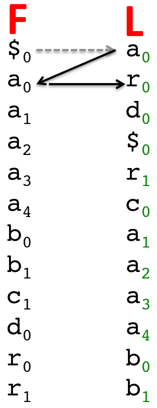
F	L
$\$_0$	a_0
a_0	r_0
a_1	d_0
a_2	$\$_0$
a_3	r_1
a_4	c_0
b_0	a_1
b_1	a_2
c_1	a_3
d_0	a_4
r_0	b_0
r_1	b_1

Reconstruction to date

T=... $

Reversing the BWT

- Because of the **cumulative index property** and because a come right after $, we go to the second row of the BWM and find a_0.
- The character that precedes it in T is now in the last column (**L**)

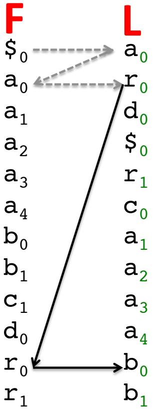


Reconstruction to date

T= … $r_0 a_0 \$$

Reversing the BWT

- To find the position of r_0 in the first column, we note that its index must be $1+5+2+1+1=10$ because of the **cumulative index property**
- We go to column **L** to get the next preceding character

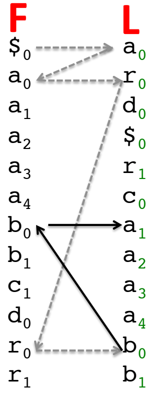

F
L
$\$_0$ a_0 a_1 a_2 a_3 a_4 b_0 b_1 c_1 d_0 r_0 r_1
a_0 r_0 d_0 $\$_0$ r_1 c_0 a_1 a_2 a_3 a_4 b_0 b_1

Reconstruction to date

T= … $b_0r_0a_0\$$

Reversing the BWT

- The game continues...
- To find the position of b_0 in the first column, we note that its index must be 1+5=6 because of the **cumulative index property**
- We go to column **L** to get the next preceding character

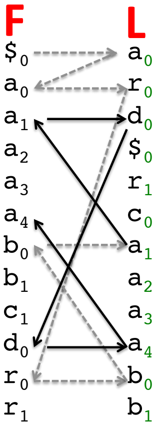


Reconstruction to date

T= ... $a_1 b_0 r_0 a_0 \$$

Reversing the BWT

- Note that to find the position of a_4 with the **cumulative index property** we take into account of the indixes of the preceding characters (i.e., \$), as well as that of a_0, a_1, a_2, a_3, so that our index is 1-4=5
- and so on...



Reconstruction to date

T= ... $a_4 d_0 a_1 b_0 r_0 a_0 \$$

Reversing the BWT

But what information exactly did we need to do this reversal?

- We can do everything starting **only** from the BWT(T)
- If we count the number of each character in BWT(T)[1], we can easily reconstruct the "chunks" of characters in the first column of the BWM

[1] Or we can store it in an array of size $\mathcal{O}(|\Sigma|)$ for characters in some alphabet Σ.

Outline

FM Index

The FM index uses the BWT and some other auxilliary data structures to generate a fast an efficient index for search for patterns within a larger string T

Paolo Ferragina and Giovanni Manzini (2000) Opportunistic Data Structures with Applications. *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. p.390.

FM Index

- The main data structures of the FM index are **F** and **L** from the BWM
- Note that **F** can be represented as an array of ints (one per character of our alphabet)
- In our example, and using the order
  $\$ < a < b < c < d < r$
  we have

1	5	2	1	1	2

- As mentioned, **L** is also easily compressible

F		L
$\$_0$	abracadabr	a_0
a_0	\$abracadab	r_0
a_1	bra\$abraca	d_0
a_2	bracadabra	$\$_0$
a_3	cadabra\$ab	r_1
a_4	dabra\$abra	c_0
b_0	ra\$abracad	a_1
b_1	racadabra\$	a_2
c_0	adabra\$abr	a_3
d_0	abra\$abrac	a_4
r_0	a\$abracada	b_0
r_1	acadabra\$a	b_1

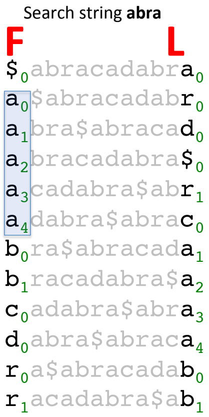
FM Index

- But how can we search?
- As mentioned, the BWM is very similar to a suffix array, but a binary search over just **F** and **L** is obviously not possible (the "middle" of the matrix is missing)
- We will again make use of the B-ranks

$$\begin{array}{lll}
\textbf{F} & & \textbf{L} \\
\mathtt{\$}_0 & \mathtt{abracadabr} & \mathtt{a}_0 \\
\mathtt{a}_0 & \mathtt{\$abracadab} & \mathtt{r}_0 \\
\mathtt{a}_1 & \mathtt{bra\$abraca} & \mathtt{d}_0 \\
\mathtt{a}_2 & \mathtt{bracadabra} & \mathtt{\$}_0 \\
\mathtt{a}_3 & \mathtt{cadabra\$ab} & \mathtt{r}_1 \\
\mathtt{a}_4 & \mathtt{dabra\$abra} & \mathtt{c}_0 \\
\mathtt{b}_0 & \mathtt{ra\$abracad} & \mathtt{a}_1 \\
\mathtt{b}_1 & \mathtt{racadabra\$} & \mathtt{a}_2 \\
\mathtt{c}_0 & \mathtt{adabra\$abr} & \mathtt{a}_3 \\
\mathtt{d}_0 & \mathtt{abra\$abrac} & \mathtt{a}_4 \\
\mathtt{r}_0 & \mathtt{a\$abracada} & \mathtt{b}_0 \\
\mathtt{r}_1 & \mathtt{acadabra\$a} & \mathtt{b}_1
\end{array}$$

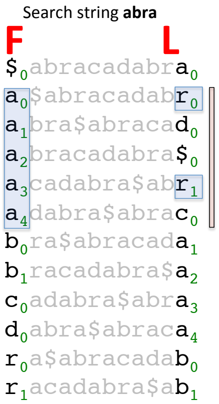
FM Index

- For example, let us search for the string `P=abra` in our "genome" `T=abracadabra`
- Our strategy is to look for all rows of BWM(T) that have P as a prefix
- We successively look for the longer P suffixes, starting with the last character of P
- But it is easy to find the chunk of the BWM(T) that starts with a given character using the **cumulative index property**



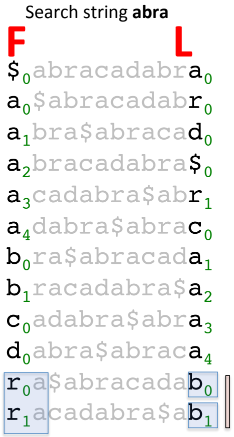
FM Index

- Once we have found all rows that begin with the last letter of P, we can look in **L** to identify those rows whose next to last letter also corresponds to P
- We can also read off the B-ranks of these characters and use the LF mapping to find the rows in **F** that begin with these characters



FM Index

- Using the LF mapping we find the rows in **F** that begin with *ra* (r_0 and r_1)
- The character that precedes “r” in our query string P is “b”, so we can continue
- We have now matched the last 3 characters of P=`abra` and continue one more step using the LF mapping


Search string **abra**

F L

$\$_0$abracadabr$a_0$

a_0\$abracadabr_0

$a_1$bra\$abracad_0

$a_2$bracadabra$\$_0$

$a_3$cadabra\$abr_1

$a_4$dabra\$abrac_0

$b_0$ra\$abracada_1

$b_1$racadabra\$$a_2$

$c_0$adabra\$abra_3

$d_0$abra\$abraca_4

r_0a\$abracadab_0

$r_1$acadabra\$ab_1

FM Index

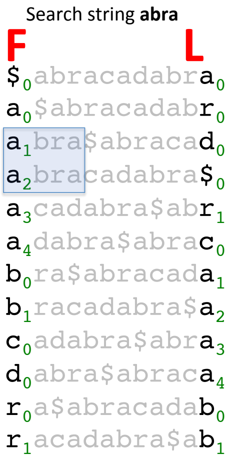
- We find the rows that begin with bra (b_0 and b_1) and look at the corresponding characters in **L** to see if we have a match for P

Search string **abra**

F		L
$\$_0$	abracadabr	a_0
a_0	\$abracadab	r_0
a_1	bra\$abraca	d_0
a_2	bracadabra	$\$_0$
a_3	cadabra\$ab	r_1
a_4	dabra\$abra	c_0
b_0	ra\$abracad	a_1
b_1	racadabra\$	a_2
c_0	adabra\$abr	a_3
d_0	abra\$abrac	a_4
r_0	a\$abracada	b_0
r_1	acadabra\$a	b_1

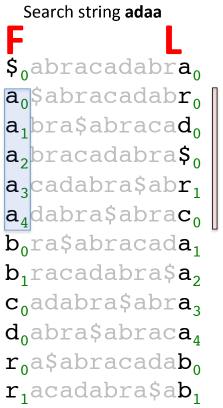
FM Index

- Finally, we find the rows of the BWM that begin with our query string: $[2,4)$
- These are equivalent to the rows we would have identified with a binary search over the suffix array (which is of course an array of start positions of suffixes)
- However, it is not immediately clear how to identify the positions in `T` that correspond to `P` using the FM index.



FM Index

- What about the search pattern `P=adaa`?
- We match the last character as previously
- But: when we now look at the corresponding rows of **L**, there is no “a”
- Ergo, the search pattern does not occur in `T`.



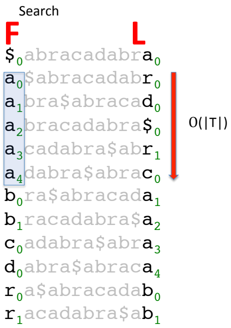
FM Index- Interim Report

We have presented a somewhat naive version of the FM index search. However, we have glossed over three issues that need to be solved to produce an efficient and practical algorithm

FM Index- Interim Report

Issue #1

- How do we efficiently find the preceding character (i.e., starting from a chunk of prefixes in or starting in **F**, how do we find the correct characters in L to continue leftwards)?
- In the worst case, we may have to scan down as far as the length of the entire input string, $\mathcal{O}(|T|)$



FM Index- Interim Report

Issue #2

- Recall that we did not want to **explicity** store the B-ranks of the characters – this would be at least 4 bytes per input character, and whatever advantage we had with respect to the suffix array would disappear
- So, we still need a way of getting the B-rank of the characters in **L**

F **L**

$\$_0$abracadabr$a_0$
a_0\$abracadabr_0
$a_1$bra\$abracad_0
$a_2$bracadabra$\$_0$
$a_3$cadabra\$abr_1
$a_4$dabra\$abrac_0
$b_0$ra\$abracada_1
$b_1$racadabra\$$a_2$
$c_0$adabra\$abra_3
$d_0$abra\$abraca_4
r_0a\$abracadab_0
$r_1$acadabra\$ab_1

FM Index- Interim Report

Issue #3

- Recall that with the suffix array, we explicitly stored the start position of each suffix of `T`
- We do not have this information with the BWM
- So, we still need a way of figuring out where matches occur in `T`

```
0:   MISSISSIPPI$                         11: $
1:   ISSISSIPPI$                          10: I$
2:   SSISSIPPI$                           7:  IPPI$
3:   SISSIPPI$                            4:  ISSIPPI$
4:   ISSIPPI$             sort            1:  ISSISSIPPI$
5:   SSIPPI$         ------------->       0:  MISSISSIPPI$
6:   SIPPI$                               9:  PI$
7:   IPPI$                                8:  PPI$
8:   PPI$                                 6:  SIPPI$
9:   PI$                                  3:  SISSIPPI$
10:  I$                                   5:  SSIPPI$
11:  $                                    2:  SSISSIPPI$
```

FM Index- Tally Table

Issue #1: efficiently find the preceding character

- Keep a tally table
- Precalculate the number of each specific character in **L** up to every row

F L	a	b	c	d	r
$\$_0 a_0$	1	0	0	0	0
$a_0 r_0$	1	0	0	0	1
$a_1 d_0$	1	0	0	1	1
$a_2 \$_0$	1	0	0	1	1
$a_3 r_1$	1	0	0	1	2
$a_4 c_0$	1	0	1	1	2
$b_0 a_1$	2	0	1	1	2
$b_1 a_2$	3	0	1	1	2
$c_0 a_3$	4	0	1	1	2
$d_0 a_4$	5	0	1	1	2
$r_0 b_0$	5	1	1	1	2
$r_1 b_1$	5	2	1	1	2

Tally table

FM Index- Tally Table

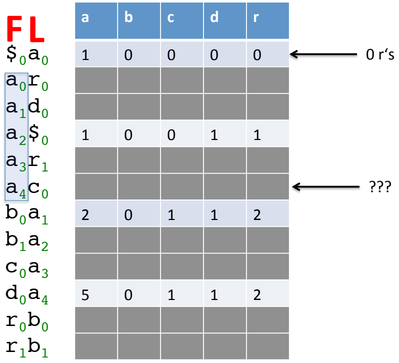
- Say we are search for `P=abra`
- After we have found all rows beginning with *a* in the first step, we need to find rows with *r* in the last column
- Say the range of rows is $[i, j]$
- We look in the tally table in row $i - 1$. No occurences of *r* to date!
- Now look in the tally table row *j*. Two occurences of *r* to date!
- Therefore, we know that (only) r_0 and r_1 occur in **L** in the range $[i, j]$

F L	a	b	c	d	r	
$\$_0 a_0$	1	0	0	0	0	← 0 r's
$a_0 r_0$	1	0	0	0	1	
$a_1 d_0$	1	0	0	1	1	
$a_2 \$_0$	1	0	0	1	1	
$a_3 r_1$	1	0	0	1	2	
$a_4 c_0$	1	0	1	1	2	← 2 r's
$b_0 a_1$	2	0	1	1	2	
$b_1 a_2$	3	0	1	1	2	
$c_0 a_3$	4	0	1	1	2	
$d_0 a_4$	5	0	1	1	2	
$r_0 b_0$	5	1	1	1	2	
$r_1 b_1$	5	2	1	1	2	

Tally table

FM Index- Tally Table

- A problem with this idea is that we need to store $\mathcal{O}(|T| \cdot |\Sigma|)$ integers
- What if we store only every k^{th} row?
- We reduce the size of the tally table by a factor of k, but at the price of not having all of the information we need immediately available

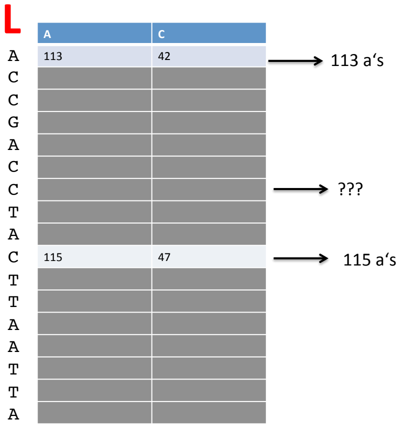


F	L	a	b	c	d	r
$\$_0$	a_0	1	0	0	0	0
a_0	r_0					
a_1	d_0					
a_2	$\$_0$	1	0	0	1	1
a_3	r_1					
a_4	c_0					
b_0	a_1	2	0	1	1	2
b_1	a_2					
c_0	a_3					
d_0	a_4	5	0	1	1	2
r_0	b_0					
r_1	b_1					

Tally table

FM Index- Tally Table

- For instance, to calculate the rank of the a near the $\leftarrow$???
- We can go to the previous checkpoint and count the number of a’s that we encounter fromthere to the position we are interested in: $113 + 1 = 114$
- Or: We can go to the next checkpoint and substract the number of a’s that we encounter along the way: $115 - 1 = 114$
- In general, we will substract one from the tally to obtain the **zero-based B-rank**



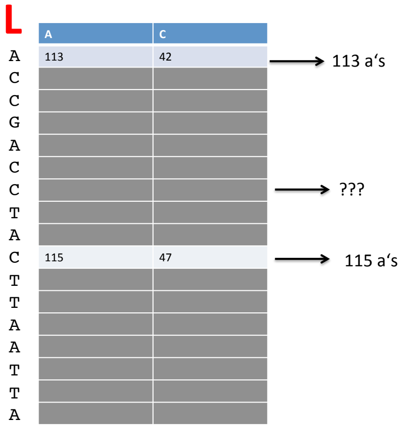
L	A	C
A	113	42
C		
C		
G		
A		
C		
C		
T		
A		
C	115	47
T		
T		
A		
A		
T		
T		
A		

113 a‘s
???
115 a‘s

Tally table

FM Index- Tally Table

- Assuming we space the check point rows a constant number of rows away from one another: $\mathcal{O}(1)$, for instance, 50 rows, then lookups are still $\mathcal{O}(1)$ rather than $\mathcal{O}(|T|)$
- We now also have a way of getting the B-ranks we need for issue # 2 (Still $\mathcal{O}(|T|)$ space, but with a smaller constant).



L	A	C
A	113	42
C		
C		
G		
A		
C		
C		
T		
A		
C	115	47
T		
T		
A		
A		
T		
T		
A		

Tally table

FM Index- Finding indices in T

F		L		
$\$_0$	abracadabr	a_0	[11]	\$
a_0	\$abracadab	r_0	[10]	a\$
a_1	bra\$abraca	d_0	[7]	abra\$
a_2	bracadabra	$\$_0$	[0]	abracadabra\$
a_3	cadabra\$ab	r_1	[3]	acadabra\$
a_4	dabra\$abra	c_0	[5]	adabra\$
b_0	ra\$abracad	a_1	[8]	bra\$
b_1	racadabra\$	a_2	[1]	bracadabra\$
c_0	adabra\$abr	a_3	[4]	cadabra\$
d_0	abra\$abrac	a_4	[6]	dabra\$
r_0	a\$abracada	b_0	[9]	ra\$
r_1	acadabra\$a	b_1	[2]	racadabra\$

- Issue #3 referred to the desire to have information as in the suffix array that would allow us to find the position of matches in the original string
- Recall the suffix array stores the indices of suffixes that are equivalent to the strings of the BWM

FM Index- Finding indices in T

F		L		
$\$_0$	abracadabr	a_0	[11]	$
a_0	$abracadab	r_0	[10]	a$
a_1	bra$abraca	d_0	[7]	abra$
a_2	bracadabra	$\$_0$	[0]	abracadabra$
a_3	cadabra$ab	r_1	[3]	acadabra$
a_4	dabra$abra	c_0	[5]	adabra$
b_0	ra$abracad	a_1	[8]	bra$
b_1	racadabra$	a_2	[1]	bracadabra$
c_0	adabra$abr	a_3	[4]	cadabra$
d_0	abra$abrac	a_4	[6]	dabra$
r_0	a$abracada	b_0	[9]	ra$
r_1	acadabra$a	b_1	[2]	racadabra$

```
abracadabra$
abra            Pos. 0
       abra     Pos. 7
```

- For instance, if we had just used the algorithm described above to find two occurences of the pattern abra then we could look up the start positions 0 and 7 if we also had the suffix array

FM Index- Finding indices in T

F		L		
$\$_0$	abracadabr	a_0	[11]	$
a_0	$abracadab	r_0	[10]	a$
a_1	bra$abracad	d_0	[7]	abra$
a_2	bracadabra	$\$_0$	[0]	abracadabra$
a_3	cadabra$ab	r_1	[3]	acadabra$
a_4	dabra$abra	c_0	[5]	adabra$
b_0	ra$abracad	a_1	[8]	bra$
b_1	racadabra$	a_2	[1]	bracadabra$
c_0	adabra$abr	a_3	[4]	cadabra$
d_0	abra$abrac	a_4	[6]	dabra$
r_0	a$abracada	b_0	[9]	ra$
r_1	acadabra$a	b_1	[2]	racadabra$

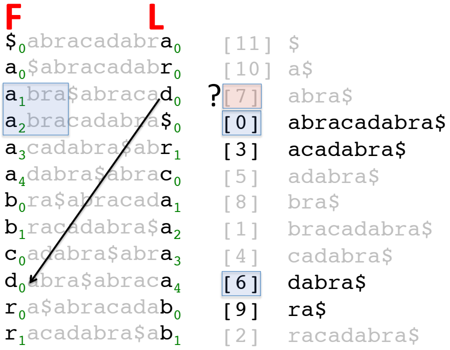
- But, if we stored the entire suffix array, this would incur roughly an additional $4 \times |\mathtt{T}|$ bytes of storage
- We can use the same checkpoint idea
- Don't store all of the values of the suffix area, just store every k^{th} value
- Importantly, we store every k^{th} value for the original string T, not every kth value in the original suffix array – this ensures constant time.

FM Index- Finding indices in T

F		L			
$\$_0$	abracadabr	a_0		[11]	$
a_0	$abracadab	r_0		[10]	a$
a_1	bra$abracad	d_0	?	[7]	abra$
a_2	bracadabra	$\$_0$		[0]	abracadabra$
a_3	cadabra$abr	r_1		[3]	acadabra$
a_4	dabra$abra	c_0		[5]	adabra$
b_0	ra$abracad	a_1		[8]	bra$
b_1	racadabra$	a_2		[1]	bracadabra$
c_0	adabra$abr	a_3		[4]	cadabra$
d_0	abra$abrac	a_4		[6]	dabra$
r_0	a$abracada	b_0		[9]	ra$
r_1	acadabra$a	b_1		[2]	racadabra$

- So, let's again search for the pattern P=abra
- We find one hit and our "selective suffix array" indicates the index to be at position 0
- What do we do about the other hit?

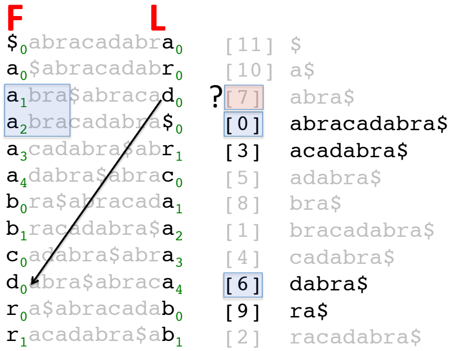
FM Index- Finding indices in T



F		L		
$\$_0$	abracadabr	a_0	[11]	$
a_0	$abracadab	r_0	[10]	a$
a_1	bra$abraca	d_0	?[7]	abra$
a_2	bracadabra	$\$_0$	[0]	abracadabra$
a_3	cadabra$ab	r_1	[3]	acadabra$
a_4	dabra$abra	c_0	[5]	adabra$
b_0	ra$abracad	a_1	[8]	bra$
b_1	racadabra$	a_2	[1]	bracadabra$
c_0	adabra$abr	a_3	[4]	cadabra$
d_0	abra$abrac	a_4	[6]	dabra$
r_0	a$abracada	b_0	[9]	ra$
r_1	acadabra$a	b_1	[2]	racadabra$

- Let us take advantage of the LF mapping
- This tells us where to find the d_0 in the first column **F**
- We can look this up in our selective suffix array – but note that we have moved one position to the left – the position of dabra is 6, but the position of abra is 7!

FM Index- Finding indices in T



- Note that the fact that we are storing every k^{th} value for the original string T, ensures that we need to perform at most $k-1$ “hops” to retrieve the index we are looking for
- However, we are still keeping $\mathcal{O}(|T|)$ elements in the selective suffix array

FM Index- Memory footprint

The FM index has a substantially smaller memory footprint than does the suffix tree (at least 60 GB) or the suffix array (at least 12 GB)

Component	Complexity	Size (Human Genome)
F	$\mathcal{O}(\|\Sigma\|)$	16 bytes (4 ints)
L	$\|T\|$ chars	2 bits $\times 3 \times 10y \approx 750$ MB
selective SA	$\sim \frac{1}{k}\|T\|$ integers	400 MB with $k = 32$
checkpoints	$\sim \frac{1}{x}\|T\| \cdot \|\Sigma\|$ integers	100 MB with $x = 128$

- Total size for FM index of human genome thus about 1.5 GB

Notes: (i) We store the 4 nucleotides with 2 bits each, i.e., 4 nucleotides per byte. (ii) k and x are the lengths of the skips

Outline

BWT/FM Index algorithms for read mapping

There are lots of published aligners for genomic resequencing. Perhaps the best known amongst them use the BWT/FM Index plus lots of **Bells and Whistles**.

- **bwa**: Li H, Durbin R (2009) Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics* **25**:1754-60.
- **bowtie**: Langmead B, Trapnell C, Pop M, Salzberg SL (2009) Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol* **10**:R25.
- **SOAP2**: Li R et al (2009) SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*. **25**:1966-7.
- . . .

For no particular reason, I will concentrate on bwa for the rest of today

bwa

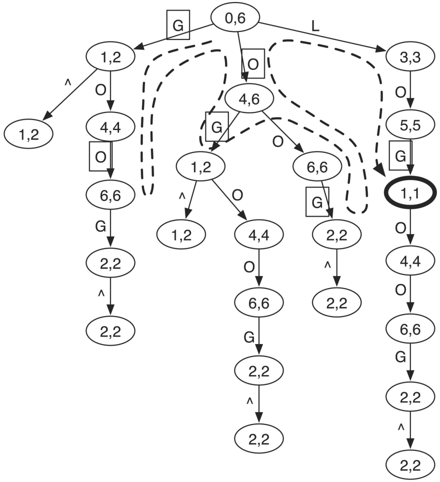
The nomenclature and descriptions used in the bwa paper are different in a few ways to those used in this lecture.

- Here I will present some of the aspects of the paper
- Exact matching is performed roughly as described
- A major issue that needs to be solved by any practical read mapper is **inexact matching**
- We will introduce the topic of inexact matching with the brute force approach that is mentioned (and rejected) in the introduction to the bwa paper

bwa: GOOGOL

The prefix trie for string X is a tree where each edge is labeled with a symbol and the string concatenation of the edge symbols on the path from a leaf to the root gives a unique prefix of X.

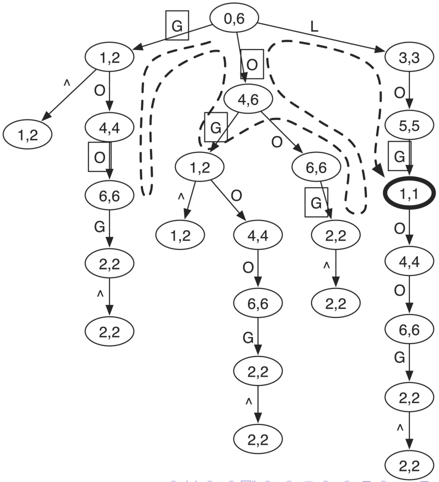
On the prefix trie, the string concatenation of the edge symbols from a node to the root gives a unique substring of X, called the string represented by the node.



bwa: GOOGOL

Note that the prefix trie of T is **identical to the suffix trie of the reverse of T**

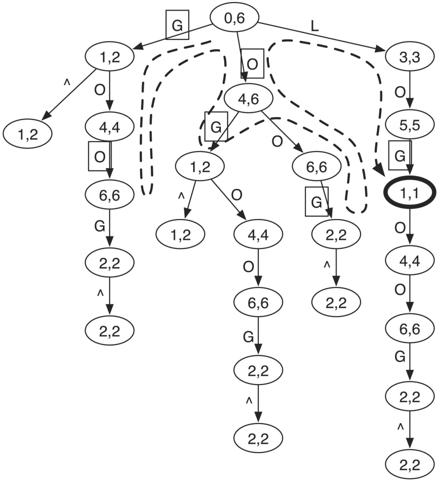
With the prefix trie, testing whether a query W is an exact substring of T is equivalent to finding the node that represents W, which can be done in $\mathcal{O}(|W|)$ time by matching each symbol in W to an edge, starting from the root.



bwa: GOOGOL

Consider the suffix array and the prefix trie of GOOGOL

0	6	$googo	l
1	3	gol$go	o
2	0	googol	$
3	5	l$goog	o
4	2	ogol$g	o
5	4	ol$goo	g
6	1	oogol$	g



Symbol ∧ marks the start of the string. The two numbers in a node give the SA interval of the string represented by the node

bwa: GOOGOL

bwa uses the following notation for **"suffix array interval"**

- All occurrences of subsrings with a common suffix W appear next to each other in the suffix array, defining the SA interval

$$[\underline{R}(W), \overline{R}(W)]$$

- For instance, the SA interval of "go" is $[1, 2]$ and the suffix array interval of "o" is $[4, 6]$

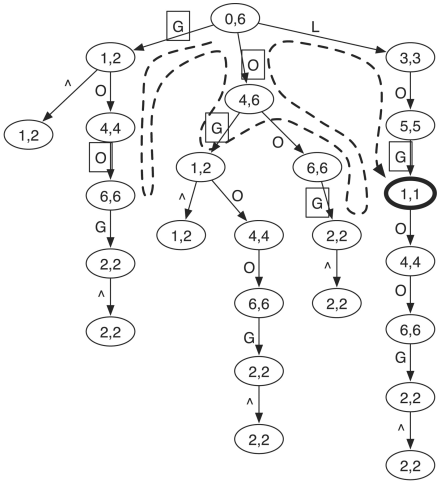
0	6	$googo	l
1	3	gol$go	o
2	0	googol	$
3	5	l$goog	o
4	2	ogol$g	o
5	4	ol$goo	g
6	1	oogol$	g

bwa: GOOGOL

Consider the suffix array and the prefix trie of GOOGOL

0	6	$googo	l
1	3	gol$go	o
2	0	googol	$
3	5	l$goog	o
4	2	ogol$g	o
5	4	ol$goo	g
6	1	oogol$	g

The dashed line shows the route of the brute-force search for a query string **LOL**, allowing at most one mismatch. Edge labels in squares mark the mismatches to the query in searching. The only hit is the bold node [1, 1] which represents string **GOL**.



bwa: GOOGOL

bwa uses the following notation for “suffix array interval”

The bwa paper presents our method of calculating the SA interval of the query W using a slightly different notation

- Can be done iteratively from the end of W

$$\begin{aligned}\underline{R}(aW) &= C(a) + Occ(a, \underline{R}(W) - 1) + 1 \\ \overline{R}(aW) &= C(a) + Occ(a, \underline{R}(W))\end{aligned}$$

where

- $C(a)$ = Number of symbols in $X[0, n-2]$ that are lexicographically smaller than a
- $Occ(a, i)$= Number of occurrences of a in $BWT[0, i]$

0	6	$googo	l
1	3	gol$go	o
2	0	googol	$
3	5	l$goog	o
4	2	ogol$g	o
5	4	ol$goo	g
6	1	oogol$	g

bwa: Inexact matching, precalculations (1)

Let us follow along the example in the bwa paper (Figure 1 and Figure 3). We have

- Reference string X = 'GOOGOL$'
- Query string W = 'LOL'
- The precalculations require us to calculate the BWT(X). For convenience, we show the sorted BWM

```
0: $GOOGOL
1: GOL$GOO
2: GOOGOL$
3: L$GOOGO
4: OGOL$GO
5: OL$GOOG
6: OOGOL$G
```

- The BWT(X)='LO$OOGG'

bwa: Inexact matching, precalculations (2)

- We now calculate $C(a)$ for X = 'GOOGOL$', defined in the paper as the number of symbols in $X[0, n2]$ that are lexicographically smaller than $a \in \Sigma$
- Let us assume $\Sigma = \{G, L, O\}$
- The vector **C** is then

a	$C(a)$
G	0
L	2
O	5

bwa: Inexact matching, precalculations (3)

- We now calculate $O(a, i)$ the number of occurrences of a in $B[0, i]$, where B is the BWT of X

i	a	$O(G, i)$	$O(L, i)$	$O(O, i)$
0	G	1	0	0
1	O	1	0	1
2	O	1	0	2
3	G	2	0	2
4	O	2	0	3
5	L	2	1	3

bwa: Inexact matching

The overall algorithm looks like this

Algorithm 2 InexactSearch(W, z)

```
1: CalculateD(W)
2: return InexRecur($W, |W| - 1, z, 1, |X| - 1$)
```

- InexRecur(W, i, z, k, l) returns the SA intervals of substrings in X that match W with no more than z differences
  - W: query
  - i: Search for matches to $W[0..i]$
  - z max number of mismatches
  - k, l: On the condition that the suffix W_{i+1} matches interval $[k..l]$

bwa: Inexact matching

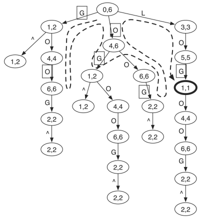
Let us examine the `CalculateD(W)` algorithm

Algorithm 3 CalculateD(W)

```
 1: $z \leftarrow 0$
 2: $j \leftarrow 0$
 3: for $i = 0$ to $|W| - 1$ do
 4:     if $W[j..i]$ is not a substring of $X$ then
 5:         $z \leftarrow z + 1$
 6:         $j \leftarrow i + 1$
 7:     end if
 8:     $D(i) \leftarrow z$
 9: end for
10: return D
```

$D(i)$ is the **lower bound** of the number of differences in $W[0..i]$ to the best match in X

bwa: Inexact matching



- Consider that we can implement a search for inexact matches as a depth-first search (as shown here) or as a breadth first search (which is actually what bwa does)
- We can bound the DFS if we know that it does not make any sense to continue the search. `CalculateD(W)` is a heuristic that allows us to **stop** the DFS **early**

bwa: Inexact matching

Algorithm 4 CalculateD(W)

```
1: $z \leftarrow 0$
2: $j \leftarrow 0$
3: for $i = 0$ to $|W| - 1$ do
4:     if $W[j..i]$ is not a substring of $X$ then
5:         $z \leftarrow z + 1$
6:         $j \leftarrow i + 1$
7:     end if
8:     $D(i) \leftarrow z$
9: end for
10: return D
```

- For X = 'GOOGOL$' and W='LOL', the for loop goes from 0..2
- we obtain D(0)=0, D(1)=1, D(2)=1

bwa: Inexact matching

Algorithm 5 InexRecur(W, i, z, k, ℓ)

```
1: if $z < D(i)$ then
2:     return $\emptyset$
3: end if
4: if $i < 0$ then
5:     return $\{k, \ell\}$    //i.e., an SA interval
6: end if
7: $I \leftarrow \emptyset$
8: for each $b \in \{a, c, g, t\}$ do
9:     $k \leftarrow C(b) + O(b, k-1) + 1$
10:    $\ell \leftarrow C(b) + O(b, \ell) + 1$
11:    if $k \leq \ell$ then
12:        if $b = W[i]$ then
13:            $I \leftarrow I \cup \mathrm{InexRecur}(W, i-1, z, k, \ell)$    //match
14:        else
15:            $I \leftarrow I \cup \mathrm{InexRecur}(W, i-1, z-1, k, \ell)$    //mismatch, decrement $z$
16:        end if
17:    end if
18: end for
19: return D
```

Lines 1-3

- If the lower bound on the number of differences in $W[0..i]$ is already more than the maximum number of mismatches z, give up
- return null

bwa: Inexact matching

Algorithm 6 InexRecur(W, i, z, k, ℓ)

```
1: if $z < D(i)$ then
2:     return $\emptyset$
3: end if
4: if $i < 0$ then
5:     return $\{k, \ell\}$    //i.e., an SA interval
6: end if
7: $I \leftarrow \emptyset$
8: for each $b \in \{a, c, g, t\}$ do
9:     $k \leftarrow C(b) + O(b, k-1) + 1$
10:    $\ell \leftarrow C(b) + O(b, \ell) + 1$
11:    if $k \leq \ell$ then
12:        if $b = W[i]$ then
13:            $I \leftarrow I \cup \mathrm{InexRecur}(W, i-1, z, k, \ell)$    //match
14:        else
15:            $I \leftarrow I \cup \mathrm{InexRecur}(W, i-1, z-1, k, \ell)$    //mismatch, decrement $z$
16:        end if
17:    end if
18: end for
19: return D
```

Lines 4-5

- If $i < 0$ then we are arriving from a recursive call where we have finished matching W (potentially including up to z mismatches)
- We return the SA interval $\{k, \ell\}$ representing the hits

bwa: Inexact matching

Algorithm 7 InexRecur(W, i, z, k, ℓ)

```
1: if $z < D(i)$ then
2:     return $\emptyset$
3: end if
4: if $i < 0$ then
5:     return $\{k, \ell\}$     //i.e., an SA interval
6: end if
7: $I \leftarrow \emptyset$
8: for each $b \in \{a, c, g, t\}$ do
9:     $k \leftarrow C(b) + O(b, k-1) + 1$
10:    $\ell \leftarrow C(b) + O(b, \ell) + 1$
11:    if $k \leq \ell$ then
12:        if $b = W[i]$ then
13:            $I \leftarrow I \cup \mathrm{InexRecur}(W, i-1, z, k, \ell)$     //match
14:        else
15:            $I \leftarrow I \cup \mathrm{InexRecur}(W, i-1, z-1, k, \ell)$     //mismatch, decrement $z$
16:        end if
17:    end if
18: end for
19: return D
```

Line 7

- Initialize the **current interval** to the empty set for this recursion

bwa: Inexact matching

Algorithm 8 InexRecur(W, i, z, k, ℓ)

```
1: if $z < D(i)$ then
2:     return $\emptyset$
3: end if
4: if $i < 0$ then
5:     return $\{k, \ell\}$    //i.e., an SA interval
6: end if
7: $I \leftarrow \emptyset$
8: for each $b \in \{a, c, g, t\}$ do
9:     $k \leftarrow C(b) + O(b, k-1) + 1$
10:    $\ell \leftarrow C(b) + O(b, \ell) + 1$
11:    if $k \leq \ell$ then
12:        if $b = W[i]$ then
13:            $I \leftarrow I \cup \mathrm{InexRecur}(W, i-1, z, k, \ell)$    //match
14:        else
15:            $I \leftarrow I \cup \mathrm{InexRecur}(W, i-1, z-1, k, \ell)$    //mismatch, decrement $z$
16:        end if
17:    end if
18: end for
19: return D
```

Line 8

- loop over all nucleotides, looking for a match...

bwa: Inexact matching

Algorithm 9 InexRecur(W, i, z, k, ℓ)

```
1: if $z < D(i)$ then
2:     return $\emptyset$
3: end if
4: if $i < 0$ then
5:     return $\{k, \ell\}$    //i.e., an SA interval
6: end if
7: $I \leftarrow \emptyset$
8: for each $b \in \{a, c, g, t\}$ do
9:     $k \leftarrow C(b) + O(b, k-1) + 1$
10:    $\ell \leftarrow C(b) + O(b, \ell) + 1$
11:    if $k \leq \ell$ then
12:        if $b = W[i]$ then
13:            $I \leftarrow I \cup \mathrm{InexRecur}(W, i-1, z, k, \ell)$    //match
14:        else
15:            $I \leftarrow I \cup \mathrm{InexRecur}(W, i-1, z-1, k, \ell)$    //mismatch, decrement $z$
16:        end if
17:    end if
18: end for
19: return D
```

Lines 9–11

- Figure out the interval in **F** where the current character b would be
- check whether this interval is empty

bwa: Inexact matching

Algorithm 10 InexRecur(W, i, z, k, ℓ)

```
1: if $z < D(i)$ then
2:     return $\emptyset$
3: end if
4: if $i < 0$ then
5:     return $\{k, \ell\}$    //i.e., an SA interval
6: end if
7: $I \leftarrow \emptyset$
8: for each $b \in \{a, c, g, t\}$ do
9:     $k \leftarrow C(b) + O(b, k-1) + 1$
10:    $\ell \leftarrow C(b) + O(b, \ell) + 1$
11:    if $k \leq \ell$ then
12:        if $b = W[i]$ then
13:            $I \leftarrow I \cup \mathrm{InexRecur}(W, i-1, z, k, \ell)$    //match
14:        else
15:            $I \leftarrow I \cup \mathrm{InexRecur}(W, i-1, z-1, k, \ell)$    //mismatch, decrement $z$
16:        end if
17:    end if
18: end for
19: return D
```

Lines 12–15

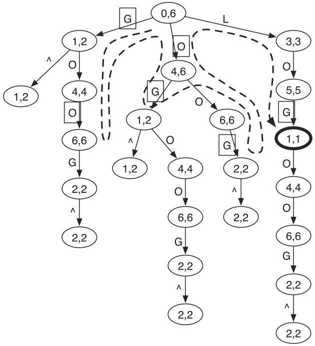
- If we have a match, keep going and decrement i
- If we have a mismatch, then also decrement z and keep going

bwa: Inexact matching

Consider now the role of the D matrix in the DFS shown in the figure

- The initial call to InexRecur($W, i-1, z-1, k, \ell$) (with W=LOL and X=GOOGOL\$ and maximally one mismatch allowed) is
- `InexRecur`$(W, |W|-1, z, 1, |X|-1)$ i.e., `InexRecur`$(W, 2, 1, 1, 6)$
- The DFS first passes by lines 1–7 from the root node and chooses the character 'G'
- G does not match the fiurst character of 'LOL', so there is a mismatch, and we recursively call `InexRecur`
- The recursive call looks like this `InexRecur`$(W, 1, 0, 1, 6)$
- When we get to line 1, $i = 1$ and $z = 0$. Recalling that we calculated $D(1) = 1$, we have that $z < D(i)$, and we return without having examined the subtree emanating from 'G'
- similarly, we avoid descending into the 'O' subtree

bwa: Inexact matching



- Therefore, our use of the D matrix allowed use to avoid continuing the DFS in two subtrees of this prefix trie

Finally

- Email: peter.robinson@charite.de
- Office hours by appointment

Further reading

- Parts of these slides were adapted from the brilliant Youtube lectures of Ben Langmead on the BWT/FM index (any infelicities are only my fault)
- Langmead B, Trapnell C, Pop M, Salzberg SL (2009) Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol* 10:R25.
- Li H, Durbin R (2009) Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics **25**:1754-60.
- Li H, Homer N (2010) A survey of sequence alignment algorithms for next-generation sequencing. *Brief Bioinform.* 11:473-83.