Exact String Matching and searching for SNPs (2)

CMSC423
The problem

• Given:
  – 100’s of millions of short reads: 100-200bp reads
  – A long reference genome (~3Bbp for human)

• Do:
  – Find high scoring (fitting) alignments for each read

• What we know:
  – Dynamic programming solution for fitting alignment:
    • $1 \times 10^8 \times 1 \times 10^9 \times 1 \times 10^2$ operations, $1 \times 10^9 \times 1 \times 10^2$ memory
Strategies

• What if we only allow a small number of substitutions?
  – Let’s first try to find *exact* matches and work from those (the d+1 trick in the midterm)

• We are aligning to the same reference 100’s of millions of time
  – Is there preprocessing we can do to amortize time?

• Genomes are repetitive
  – Can we search for matches in the genome in a smart way?
  – Can we compress the genome, and search over the compressed representation?
Suffix Tree

T: abaaba$
0123456

- Collapse non-branching nodes
  - #nodes \(O(|T|)\)

- Memory requirement is \textit{not} \(O(|T|)\)
  - In the worst case, space required for edge labels is \(O(|T|)\)
Suffix Tree

T: abaaba$
   0 1 2 3 4 5 6

- Collapse non-branching nodes
  - #nodes $O(|T|)$

- Label edges with substring $[\text{start}, \text{end}]$
  - $O(1)$ per edge

- Memory now $O(|T|)$

- Construction algorithm $O(|T|)$ (see Gusfield)
## Recap

<table>
<thead>
<tr>
<th>Structure</th>
<th>Processing Time</th>
<th>Memory</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suffix Trie</td>
<td>$O(</td>
<td>T</td>
<td>)$</td>
</tr>
<tr>
<td>Suffix Tree</td>
<td>$O(</td>
<td>T</td>
<td>)$</td>
</tr>
<tr>
<td>Suffix Array</td>
<td>$O(</td>
<td>T</td>
<td>)$</td>
</tr>
</tbody>
</table>

*In best implementations about 20 bytes per character (as opposed to 4 bytes for suffix array)*
Suffix Arrays

- Even though Suffix Trees are $O(n)$ space, the constant hidden by the big-Oh notation is somewhat “big”: $\approx 20$ bytes / character in good implementations.

- If you have a 10Gb genome, 20 bytes / character = 200Gb to store your suffix tree. “Linear” but large.

- Suffix arrays are a more efficient way to store the suffixes that can do most of what suffix trees can do, but just a bit slower.

- Slight space vs. time tradeoff.
Example Suffix Array

s = attcatg$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

<table>
<thead>
<tr>
<th>Index of Suffix</th>
<th>Suffix of s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>attcatg$</td>
</tr>
<tr>
<td>2</td>
<td>ttcatg$</td>
</tr>
<tr>
<td>3</td>
<td>tcatg$</td>
</tr>
<tr>
<td>4</td>
<td>catg$</td>
</tr>
<tr>
<td>5</td>
<td>atg$</td>
</tr>
<tr>
<td>6</td>
<td>tg$</td>
</tr>
<tr>
<td>7</td>
<td>g$</td>
</tr>
<tr>
<td>8</td>
<td>$</td>
</tr>
</tbody>
</table>

Sort the suffixes alphabetically:
- the indices just “come along for the ride”
Example Suffix Array

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

\[ s = \texttt{attcatg}$ \]

<table>
<thead>
<tr>
<th>Index</th>
<th>Suffix of s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\texttt{attcatg}$</td>
</tr>
<tr>
<td>2</td>
<td>\texttt{ttcatg}$</td>
</tr>
<tr>
<td>3</td>
<td>\texttt{tcactg}$</td>
</tr>
<tr>
<td>4</td>
<td>\texttt{catg}$</td>
</tr>
<tr>
<td>5</td>
<td>\texttt{atg}$</td>
</tr>
<tr>
<td>6</td>
<td>\texttt{tg}$</td>
</tr>
<tr>
<td>7</td>
<td>\texttt{g}$</td>
</tr>
<tr>
<td>8</td>
<td>\texttt{$}</td>
</tr>
</tbody>
</table>

sort the suffixes alphabetically

the indices just "come along for the ride"

\[ \text{index of suffix} \quad \text{suffix of s} \]

\[ \begin{array}{c|c}
1 & \texttt{attcatg}$ \\
2 & \texttt{ttcatg}$ \\
3 & \texttt{tcactg}$ \\
4 & \texttt{catg}$ \\
5 & \texttt{atg}$ \\
6 & \texttt{tg}$ \\
7 & \texttt{g}$ \\
8 & \texttt{$} \\
\end{array} \]
Search via Suffix Arrays

$s = \texttt{cattcat}$

- Does string “at” occur in $s$?
- Binary search to find “at”.
- What about “tt”?

<table>
<thead>
<tr>
<th>8</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>at$</td>
</tr>
<tr>
<td>2</td>
<td>attcat$</td>
</tr>
<tr>
<td>5</td>
<td>cat$</td>
</tr>
<tr>
<td>1</td>
<td>cattcat$</td>
</tr>
<tr>
<td>7</td>
<td>t$</td>
</tr>
<tr>
<td>4</td>
<td>tcat$</td>
</tr>
<tr>
<td>3</td>
<td>ttcat$</td>
</tr>
</tbody>
</table>
Counting via Suffix Arrays

How many times does “at” occur in the string?

All the suffixes that start with “at” will be next to each other in the array.

Find one suffix that starts with “at” (using binary search).

Then count the neighboring sequences that start with at.

s = cattcat$

| 8 | $      |
| 6 | at$    |
| 2 | attcat$|
| 5 | cat$   |
| 1 | cattcat$|
| 7 | t$     |
| 4 | tcat$  |
| 3 | ttcat$ |
Constructing Suffix Arrays

- Easy $O(n^2 \log n)$ algorithm:
  
  sort the $n$ suffixes, which takes $O(n \log n)$ comparisons, where each comparison takes $O(n)$.

- There are several direct $O(n)$ algorithms for constructing suffix arrays that use very little space.

- An simple $O(n)$ algorithm: build the suffix tree, and exploit the relationship between suffix trees and suffix arrays (next slide)
Relationship between Suffix Arrays and Suffix Trees

T: abaaba$
0123456

Build suffix trees with edge labels sorted lexicographically
Order of leaves: 6,5,2,3,0,4,1
# Recap

<table>
<thead>
<tr>
<th>Structure</th>
<th>Processing Time</th>
<th>Memory</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suffix Trie</td>
<td>O(</td>
<td>T</td>
<td>)</td>
</tr>
<tr>
<td>Suffix Tree</td>
<td>O(</td>
<td>T</td>
<td>)</td>
</tr>
<tr>
<td>Suffix Array</td>
<td>O(</td>
<td>T</td>
<td>) (but much smaller than Suffix Tree)</td>
</tr>
</tbody>
</table>

*In best implementations about 20 bytes per character (as opposed to 4 bytes for suffix array)
Burrows-Wheeler Transform

Text transform that is useful for compression & search.

\[
\text{BWT}(\text{banana}) = \text{annb$aa}
\]

Tends to put runs of the same character together.

Makes compression work well.

“bzip” is based on this.
Another Example

BWT(appellee$) = e$elplepa

Doesn’t always improve the compressibility...
Recovering the string

<table>
<thead>
<tr>
<th>BWT</th>
<th>sort BWT</th>
<th>first column</th>
<th>first 2 columns</th>
<th>first 3 columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>e $</td>
<td>$a</td>
<td>e $a</td>
<td>$ap</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>ap</td>
<td>ap</td>
<td>app</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>e$</td>
<td>e$</td>
<td>e$</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>ee</td>
<td>ee</td>
<td>ee$</td>
<td></td>
</tr>
<tr>
<td>e$</td>
<td>e$ee</td>
<td>e$ee</td>
<td>ee$</td>
<td></td>
</tr>
<tr>
<td>$appellee</td>
<td>sort these 2 columns</td>
<td>$appellee</td>
<td>$appellee</td>
<td>$appellee</td>
</tr>
<tr>
<td>appellee$</td>
<td>ee</td>
<td>ee</td>
<td>ee</td>
<td>ee</td>
</tr>
<tr>
<td>e$appelle</td>
<td>e$</td>
<td>e$</td>
<td>e$</td>
<td>e$</td>
</tr>
<tr>
<td>ee$appell</td>
<td>ee$</td>
<td>ee$</td>
<td>ee$</td>
<td>ee$</td>
</tr>
<tr>
<td>elle$appep</td>
<td>elle$</td>
<td>elle$</td>
<td>elle$</td>
<td>elle$</td>
</tr>
<tr>
<td>llee$appepe</td>
<td>llee$</td>
<td>llee$</td>
<td>llee$</td>
<td>llee$</td>
</tr>
<tr>
<td>ppelle$a</td>
<td>pp</td>
<td>pp</td>
<td>pp</td>
<td>pp</td>
</tr>
<tr>
<td>ppelle$a</td>
<td>pp</td>
<td>pp</td>
<td>pp</td>
<td>pp</td>
</tr>
<tr>
<td>app</td>
<td>pp</td>
<td>pp</td>
<td>pp</td>
<td>pp</td>
</tr>
</tbody>
</table>

Sort these 3 columns: ell, lee, lle, pel, ppe.
def inverseBWT(s):
    B = [s_1, s_2, s_3, ..., s_n]
    for i = 1..n:
        sort B
        prepend s_i to B[i]
    return row of B that ends with $
Another BWT Example

\text{dogwood$} \quad \text{ogwood$d} \quad \text{gwood$do} \quad \text{wood$dog} \quad \text{ood$dogw} \quad \text{ood$dog} \quad \text{od$dogwo} \quad \text{d$dogwoo} \quad \text{$dogwood}$

\text{dogwood$} \quad \text{ogwood$d} \quad \text{gwood$do} \quad \text{wood$dog} \quad \text{ood$dogw} \quad \text{ood$dog} \quad \text{od$dogwo} \quad \text{d$dogwoo} \quad \text{$dogwood}$

\text{sort}\quad \text{dogwood$} \quad \text{dogwood$d} \quad \text{dogwood$} \quad \text{dogwood}$

\text{BWT(dogwood$)} = \text{do$oodwg}$
Another BWT Example

Prepend | Sort | Prepend | Sort | Prepend | Sort | Prepend | Sort
---|---|---|---|---|---|---|---

d $ | $d | d $ | $d | $do | $do | $dog | $dog
odo | d$ | o d$ | d$ | d$ | o$d | d$ | o$d$ | d$ | d$ | o$d$ | d$ | do | d$ | dog | dogw
og | gw | o gw | gwo | ogwo | gwoo | $ | dogw | dogwo
oo | od | o od | od$ | ood$ | od$ | d | ood$d | od$ | do | ood$ | od$ | d | dogwo | ogwoo
wo | oo | w oo | ood | wood | ood$ | $ | wood$ | ood$d | gwood | wood
gw | wo | g wo | woo | gwoo | wood | dogwo | ogwo | $ | dogwo | wood
wood | $ | dogwo | $ | dogwo | $ | dogwood | o | d | dogwood | wood
ogwo | $ | dogwoo | $ | dogwood | $ | dogwood | o | d | dogwood | wood
ood$ | do | od$ | dog | ood$ | dog | $ | dogwood | ood$ | do | ood$ | dog | $ | dogwood | ood$ | do | ood$ | dog | $ | dogwood
Searching with BWT: LF Mapping

<table>
<thead>
<tr>
<th>BWT(unabashable)</th>
<th>$a\ b\ e\ h\ l\ n\ s\ u$</th>
<th>$\Sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$unabashable$</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>$abashable$un</td>
<td>0 0 0 1 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>$able$unabash</td>
<td>0 0 0 1 0 0 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>$ashable$unab</td>
<td>0 0 0 1 1 0 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>$bashable$una</td>
<td>0 0 1 1 1 0 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>$ble$unabasha</td>
<td>0 1 1 1 1 0 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>$e$unabashabl</td>
<td>0 2 1 1 1 0 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>$hable$unabas</td>
<td>0 2 1 1 1 1 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>$le$unabashab</td>
<td>0 2 1 1 1 1 1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>$nabashable$u</td>
<td>0 2 2 1 1 1 1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>$shable$unaba</td>
<td>0 2 2 1 1 1 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>$unabashable$</td>
<td>0 3 2 1 1 1 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 3 2 1 1 1 1 1 1 1</td>
<td></td>
</tr>
</tbody>
</table>

LF Property: The $i^{th}$ occurrence of a letter $X$ in the last column corresponds to the $i^{th}$ occurrence of $X$ in the first column.
# BWT Search

**BWTSearch(aba)**  
Start from the **end** of the pattern

<table>
<thead>
<tr>
<th>Step 1: Find the range of “a”s in the first column</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 2: Look at the same range in the last column.</td>
</tr>
<tr>
<td>Step 3: “b” is the next pattern character. Set B = the LF mapping entry for b in the first row of the range. Set E = the LF mapping entry for b in the last + 1 row of the range.</td>
</tr>
<tr>
<td>Step 4: Find the range for “b” in the first row, and use B and E to find the right subrange within the “b” range.</td>
</tr>
</tbody>
</table>

## BWT(una bashable)

<table>
<thead>
<tr>
<th>Character</th>
<th>$</th>
<th>a</th>
<th>b</th>
<th>e</th>
<th>h</th>
<th>l</th>
<th>n</th>
<th>s</th>
<th>u</th>
</tr>
</thead>
<tbody>
<tr>
<td>unabashable</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>abashable</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>able</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ashable</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bashable</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ble</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>hable</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>le</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>nabashable</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>shable</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>unabashable</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

## LF Mapping

$\Sigma$

<table>
<thead>
<tr>
<th>$\Sigma$</th>
<th>$\Sigma$</th>
<th>$\Sigma$</th>
<th>$\Sigma$</th>
<th>$\Sigma$</th>
<th>$\Sigma$</th>
<th>$\Sigma$</th>
<th>$\Sigma$</th>
<th>$\Sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
<td>$\Sigma$</td>
</tr>
</tbody>
</table>

**BWT(unabashable)**
### BWT Searching Example 2

**Pattern:** "bana"

<table>
<thead>
<tr>
<th>a</th>
<th>$a b n</th>
<th>a</th>
<th>$a b n</th>
<th>a</th>
<th>$a b n</th>
</tr>
</thead>
<tbody>
<tr>
<td>$bananna</td>
<td>0 0 0 0</td>
<td>$bananna</td>
<td>0 0 0 0</td>
<td>$bananna</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>a$banann</td>
<td>0 1 0 0</td>
<td>a$banann</td>
<td>0 1 0 0</td>
<td>a$banann</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>ananna$b</td>
<td>0 1 0 1</td>
<td>ananna$b</td>
<td>0 1 0 1</td>
<td>ananna$b</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>anna$ban</td>
<td>0 1 1 1</td>
<td>anna$ban</td>
<td>0 1 1 1</td>
<td>anna$ban</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>bananna$</td>
<td>0 1 1 2</td>
<td>bananna$</td>
<td>0 1 1 2</td>
<td>bananna$</td>
<td>0 1 1 2</td>
</tr>
<tr>
<td>na$banan</td>
<td>1 1 1 2</td>
<td>na$banan</td>
<td>1 1 1 2</td>
<td>na$banan</td>
<td>1 1 1 2</td>
</tr>
<tr>
<td>nanna$ba</td>
<td>1 1 1 3</td>
<td>nanna$ba</td>
<td>1 1 1 3</td>
<td>nanna$ba</td>
<td>1 1 1 3</td>
</tr>
<tr>
<td>nna$bana</td>
<td>1 2 1 3</td>
<td>nna$bana</td>
<td>1 2 1 3</td>
<td>nna$bana</td>
<td>1 2 1 3</td>
</tr>
<tr>
<td>(B,E) = 0, 2</td>
<td></td>
<td>(B,E) = 1, 2</td>
<td></td>
<td>(B,E) = 0, 1</td>
<td></td>
</tr>
</tbody>
</table>

### Example Steps

1. **Step 1:**
   - Pattern: "bana"
   - Current Index: 0
   - Next Index: 1 (B,E) = (0,2)

2. **Step 2:**
   - Current Index: 1
   - Next Index: 2 (B,E) = (1,2)

3. **Step 3:**
   - Current Index: 2
   - Next Index: 3 (B,E) = (0,1)

4. **Step 4:**
   - Current Index: 3
   - Next Index: 0 (B,E) = (0,2)

5. **Step 5:**
   - Current Index: 0
   - Next Index: 1 (B,E) = (1,2)

6. **Step 6:**
   - Current Index: 1
   - Next Index: 2 (B,E) = (0,1)

7. **Step 7:**
   - Current Index: 2
   - Next Index: 3 (B,E) = (1,2)

8. **Step 8:**
   - Current Index: 3
   - Next Index: 0 (B,E) = (0,2)

9. **Step 9:**
   - Current Index: 0
   - Next Index: 1 (B,E) = (1,2)

10. **Step 10:**
    - Current Index: 1
    - Next Index: 2 (B,E) = (0,1)

11. **Step 11:**
    - Current Index: 2
    - Next Index: 3 (B,E) = (1,2)

12. **Step 12:**
    - Current Index: 3
    - Next Index: 0 (B,E) = (0,2)

13. **Step 13:**
    - Current Index: 0
    - Next Index: 1 (B,E) = (1,2)

14. **Step 14:**
    - Current Index: 1
    - Next Index: 2 (B,E) = (0,1)

15. **Step 15:**
    - Current Index: 2
    - Next Index: 3 (B,E) = (1,2)

16. **Step 16:**
    - Current Index: 3
    - Next Index: 0 (B,E) = (0,2)

17. **Step 17:**
    - Current Index: 0
    - Next Index: 1 (B,E) = (1,2)

18. **Step 18:**
    - Current Index: 1
    - Next Index: 2 (B,E) = (0,1)

19. **Step 19:**
    - Current Index: 2
    - Next Index: 3 (B,E) = (1,2)

20. **Step 20:**
    - Current Index: 3
    - Next Index: 0 (B,E) = (0,2)

End of example.
BWT Searching Notes

• Don’t have to store the LF mapping. A more complex algorithm (later slides) lets you compute it in $O(1)$ time in *compressed* data on the fly with some extra storage.

• To find the range in the first column corresponding to a character:
  • Pre-compute array $C[c] = \# \text{ of occurrences in the string of characters lexicographically } < c$.
  • Then start of the “a” range, for example, is: $C[“a”] + 1$.

• Running time: $O(|\text{pattern}|)$
  • Finding the range in the first column takes $O(1)$ time using the $C$ array.
  • Updating the range takes $O(1)$ time using the LF mapping.
Relationship Between BWT and Suffix Arrays

$s = \text{appellee}\$

123456789

BWT matrix

The suffixes are obtained by deleting everything after the $\$

$s[9-1] = e$

$s[1-1] = \$$

$s[8-1] = e$

$s[7-1] = l$

$s[4-1] = p$

$s[6-1] = l$

$s[5-1] = e$

$s[3-1] = p$

$s[2-1] = a$

Suffix array (start position for the suffixes)

Suffix position - 1 = the position of the last character of the BWT matrix

($\$ is a special case)

$\text{appellee}$

\text{appellee}\$

e$\text{appelle}$

ee$\text{appell}$

ellee$\text{app}$

lee$\text{appel}$

lleee$\text{appe}$

pellee$\text{ap}$

ppellee$\text{a}$

These are still in sorted order because “$\$” comes before everything else

- subtract 1
Relationship Between BWT and Suffix Trees

- Remember: Suffix Array = suffix numbers obtained by traversing the leaf nodes of the (ordered) Suffix Tree from left to right.

- Suffix Tree $\Rightarrow$ Suffix Array $\Rightarrow$ BWT.

Ordered suffix tree for previous example:

$\Sigma = \{\$, e, l, p\}$

$s = appellee\$$

123456789

284x697
Computing BWT in $O(n)$ time

- Easy $O(n^2 \log n)$-time algorithm to compute the BWT (create and sort the BWT matrix explicitly).

- Several direct $O(n)$-time algorithms for BWT. These are space efficient.

- Also can use suffix arrays or trees:
  - Compute the suffix array, use correspondence between suffix array and BWT to output the BWT.
  - $O(n)$-time and $O(n)$-space, but the constants are large.
**Recap**

BWT useful for searching and compression.

BWT is *invertible*: given the BWT of a string, the string can be reconstructed!

BWT is computable in $O(n)$ time.

Close relationships between Suffix Trees, Suffix Arrays, and BWT:

- Suffix array = order of the suffix numbers of the suffix tree, traversed left to right
- BWT = letters at positions given by the suffix array entries - 1

Even after compression, can search string quickly.
## Recap

<table>
<thead>
<tr>
<th>Structure</th>
<th>Processing Time</th>
<th>Memory</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suffix Trie</td>
<td>$O(</td>
<td>T</td>
<td>)$</td>
</tr>
<tr>
<td>Suffix Tree</td>
<td>$O(</td>
<td>T</td>
<td>)$</td>
</tr>
<tr>
<td>Suffix Array</td>
<td>$O(</td>
<td>T</td>
<td>)$</td>
</tr>
<tr>
<td>BWT</td>
<td>$O(</td>
<td>T</td>
<td>)$</td>
</tr>
</tbody>
</table>

*In best implementations about 20 bytes per character (as opposed to 4 bytes for suffix array)

**Compressed! For human genome ~2GB
# Move-To-Front Coding

To encode a letter, use its index in the current list, and then move it to the front of the list.

\[
\sum \quad \text{do$ooodwg}
\]

<table>
<thead>
<tr>
<th>Letter</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>$dgow</td>
<td>1</td>
</tr>
<tr>
<td>d$gow</td>
<td>13</td>
</tr>
<tr>
<td>od$gw</td>
<td>132</td>
</tr>
<tr>
<td>$odgw</td>
<td>1322</td>
</tr>
<tr>
<td>o$dgw</td>
<td>13220</td>
</tr>
<tr>
<td>o$dgw</td>
<td>132202</td>
</tr>
<tr>
<td>do$gw</td>
<td>1322024</td>
</tr>
<tr>
<td>wdo$g</td>
<td>13220244</td>
</tr>
</tbody>
</table>

List with all letters from the allowed alphabet

\[\text{do$ooodwg} = \text{MTF(do$ooodwg)}\]

Benefits:

- Runs of the same letter will lead to runs of 0s.
- Common letters get small numbers, while rare letters get big numbers.
Compressing BWT Strings

Lots of possible compression schemes will benefit from preprocessing with BWT (since it tends to group runs of the same letters together).

One good scheme proposed by Ferragina & Manzini:

```
PrefixCode(rle(MTF(BWT(S))))
```

- replace runs of 0s with the count of 0s
- Huffman code that uses more bits for rare symbols
Pseudocode for CountingOccurrences in BWT w/o stored LF mapping

```
function Count(S_{bwt}, P):
    c = P[p], i = p
    sp = C[c] + 1; ep = C[c+1]

    while (sp ≤ ep) and (i ≥ 2) do
        c = P[i-1]
        sp = C[c] + Occ(c, sp-1) + 1
        ep = C[c] + Occ(c, ep)
        i = i - 1

    if ep < sp then
        return “not found”
    else
        return ep - sp + 1
```

**C[c]** = index into first column where the “c”’s begin.

**Occ(c, p)** = # of of c in the first p characters of BWT(S), aka the LF mapping.
Computing Occ in Compressed String

Break BWT(S) into blocks of length L (we will decide on a value for L later):

\[
\begin{array}{c}
\text{BWT(S)} \\
\hline
\text{BT}_1 \quad \text{BT}_2 \quad \text{BT}_3 \quad \ldots \\
\end{array}
\]

PrefixCode(rle(MTF(BWT(BT_2))))

\[
\begin{array}{c}
\text{BZ}_1 \quad \text{BZ}_2 \quad \text{BZ}_3 \quad \ldots \\
\end{array}
\]

Occ(c, p) = \# of “c” up thru p

Assumes every run of 0s is contained in a block [just for ease of explanation].

We will store some extra info for each block (and some groups of blocks) to compute Occ(c, p) quickly.
**Extra Info to Compute Occ**

**block**: store $|\Sigma|$-long array giving # of occurrences of each character up thru and including this block since the end of the last super block.

**superblock**: store $|\Sigma|$-long array giving # of occurrences of each character up thru and including this superblock.
Extra Info to Compute Occ

\[ u = \text{compressed length} \]
Choose \( L = O(\log u) \)

\( \frac{u}{L} \log L = \frac{u}{\log u} \log \log u \) total space.

**block**: store \(|\Sigma|\)-long array giving # of occurrences of each character up thru and including this block since the end of the last super block.

**superblock**: store \(|\Sigma|\)-long array giving # of occurrences of each character up thru and including this superblock
Extra Info to Compute Occ

\[ u = \text{compressed length} \]

Choose \( L = O(\log u) \)

\( \frac{u}{L} \) blocks, each array is \(|\Sigma|\log L\) long \( \Rightarrow \)

\[ \frac{u}{L} \log L = \frac{u}{\log u} \log \log u \]

total space.

\[ u = \text{compressed length} \]

Choose \( L = O(\log u) \)

\[ \frac{u}{L} \log L = \frac{u}{\log u} \log \log u \]

total space.

\[ \frac{u}{L^2} \text{ superblocks, each array is } |\Sigma|\log u \text{ long} \]

\[ \Rightarrow \frac{u}{(\log u)^2} \log u = \frac{u}{\log u} \]

total space.

\[ \text{block: store } |\Sigma|\text{-long array giving } \#	ext{ of occurrences of each character up thru and including this block since the end of the last super block.} \]

\[ \text{superblock: store } |\Sigma|\text{-long array giving } \# \text{ of occurrences of each character up thru and including this superblock} \]
Extra Info to Compute Occ

\( u = \text{compressed length} \)
Choose \( L = O(\log u) \)

\[
\begin{array}{ccccccc}
\text{BZ}_1 & \text{BZ}_2 & \text{BZ}_3 & \ldots & \text{BZ}_i & \text{block} \\
\end{array}
\]

\[
\begin{array}{ccccccc}
\text{L}^2 & \text{superblock} & \text{L}^2 \\
\end{array}
\]

\( \text{Occ}(c, p) = \# \text{ of } "c" \text{ up thru } p: \)
sum value at last superblock, value at end of previous block, but then need to handle this block.

Store an array: \( M[c, k, \text{BZ}_i, \text{MTF}_i] = \# \text{ of occurrences of } c \text{ through the } k\text{th letter of a block of type } (\text{BZ}_i, \text{MTF}_i) \).

Size: \( O(|\Sigma|L2^L|\Sigma|) = O(L2^{L'}) = O(u^c \log u) \text{ for } c < 1 \text{ (since the string is compressed)} \)