Burrows-Wheeler Transform

02-714
Slides by Carl Kingsford
Motivation - Short Read Mapping

Sequencing technologies produce millions of “reads” = a random, short substring of the genome.

If we already know the genome of one cow, we can get reads from a 2nd cow and map them onto the known cow genome.

Need to do millions of string searches in a long string.
Bowtie

Ultrafast and memory-efficient alignment of short DNA sequences to the human genome
Ben Langmead*, Cole Trapnell, Mihai Pop and Steven L Salzberg

* Corresponding author: Ben Langmead langmead@cs.umd.edu

Center for Bioinformatics and Computational Biology, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA
For all author emails, please log on.


The electronic version of this article is the complete one and can be found online at:

BWA

Fast and accurate short read alignment with Burrows–Wheeler transform
Heng Li and Richard Durbin*

* To whom correspondence should be addressed.

Received February 20, 2009.
Revision received May 6, 2009.
Accepted May 12, 2009.

## Bowtie Performance

### Varying read length using Bowtie, Maq and SOAP

<table>
<thead>
<tr>
<th>Length</th>
<th>Program</th>
<th>CPU time</th>
<th>Wall clock time</th>
<th>Peak virtual memory footprint (megabytes)</th>
<th>Bowtie speed-up</th>
<th>Reads aligned (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>36 bp</td>
<td>Bowtie</td>
<td>6 m 15 s</td>
<td>6 m 21 s</td>
<td>1,305</td>
<td>-</td>
<td>62.2</td>
</tr>
<tr>
<td></td>
<td>Maq</td>
<td>3 h 52 m 26 s</td>
<td>3 h 52 m 54 s</td>
<td>804</td>
<td>36.7x</td>
<td>65.0</td>
</tr>
<tr>
<td></td>
<td>Bowtie -v 2</td>
<td>4 m 55 s</td>
<td>5 m 00 s</td>
<td>1,138</td>
<td>-</td>
<td>55.0</td>
</tr>
<tr>
<td></td>
<td>SOAP</td>
<td>16 h 44 m 3 s</td>
<td>18 h 1 m 38 s</td>
<td>13,619</td>
<td>216x</td>
<td>55.1</td>
</tr>
<tr>
<td>50 bp</td>
<td>Bowtie</td>
<td>7 m 11 s</td>
<td>7 m 20 s</td>
<td>1,310</td>
<td>-</td>
<td>67.5</td>
</tr>
<tr>
<td></td>
<td>Maq</td>
<td>2 h 39 m 56 s</td>
<td>2 h 40 m 9 s</td>
<td>804</td>
<td>21.8x</td>
<td>67.9</td>
</tr>
<tr>
<td></td>
<td>Bowtie -v 2</td>
<td>5 m 32 s</td>
<td>5 m 46 s</td>
<td>1,138</td>
<td>-</td>
<td>56.2</td>
</tr>
<tr>
<td></td>
<td>SOAP</td>
<td>48 h 42 m 4 s</td>
<td>66 h 26 m 53 s</td>
<td>13,619</td>
<td>691x</td>
<td>56.2</td>
</tr>
<tr>
<td>76 bp</td>
<td>Bowtie</td>
<td>18 m 58 s</td>
<td>19 m 6 s</td>
<td>1,323</td>
<td>-</td>
<td>44.5</td>
</tr>
<tr>
<td></td>
<td>Maq 0.7.1</td>
<td>4 h 45 m 7 s</td>
<td>4 h 45 m 17 s</td>
<td>1,155</td>
<td>14.9x</td>
<td>44.9</td>
</tr>
<tr>
<td></td>
<td>Bowtie -v 2</td>
<td>7 m 35 s</td>
<td>7 m 40 s</td>
<td>1,138</td>
<td>-</td>
<td>31.7</td>
</tr>
</tbody>
</table>

The performance of Bowtie v0.9.6, SOAP v1.10, and Maq versions v0.6.6 and v0.7.1 on the server platform when aligning 2 M untrimmed reads from the 1,000 Genome project (National Center for Biotechnology Information Short Read Archive: SRR003084 for 36 base pairs [bp], SRR003092 for 50 bp, and SRR003196 for 76 bp). For each read length, the 2 M reads were randomly sampled from the FASTQ file downloaded from the Archive such that the average per-base error rate as measured by quality values was uniform across the three sets. All reads pass through Maq’s "catfilter". Maq v0.7.1 was used for the 76-bp reads because v0.6.6 does not support reads longer than 63 bp. SOAP is excluded from the 76-bp experiment because it does not support reads longer than 60 bp. Other experimental parameters are identical to those of the experiments in Table 1. CPU, central processing unit.

*Langmead et al. (2008)*
Burrows-Wheeler Transform

Text transform that is useful for compression & search.

banana

banana$  $banana
anana$b  a$banan
nana$ba  ana$ban
ana$ban  anana$b
na$bana  banana$
a$banan  nana$ba
$banana  na$bana

$banan  \text{sort}  a$banan
ana$ban  anana$ba
na$bana  banana$
a$banan  nana$ba
$banana  na$bana

BWT(banana) = annb$aa

Tends to put runs of the same character together.

Makes compression work well.

“bzip” is based on this.
Another Example

appellee$

appellee$ appellee$
ppellee$a appellee$
pellee$ap e$appelle
elee$app e$appelle
lleee$appe elee$ap
lee$appel lee$appel
ee$appell llee$appe
e$appelle pellee$ap
$appellee ppellee$a

BWT(appellee$) = e$elplepa

Doesn’t always improve the compressibility...

Putting runs of characters together is simple (sort the characters); the real utility of the BWT is that it is completely invertable!
## Recovering the string

<table>
<thead>
<tr>
<th>Step</th>
<th>Alphabet</th>
<th>Sorted Columns</th>
<th>First 2 Columns</th>
<th>First 3 Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e $</td>
<td>e e $</td>
<td>e $ a</td>
<td>$ a</td>
</tr>
<tr>
<td>2</td>
<td>appellee</td>
<td>$a</td>
<td>$ap</td>
<td>app</td>
</tr>
<tr>
<td>3</td>
<td>appelle</td>
<td>a</td>
<td>ap</td>
<td>app</td>
</tr>
<tr>
<td>4</td>
<td>e$appelle</td>
<td>e</td>
<td>e$ap</td>
<td>ap</td>
</tr>
<tr>
<td>5</td>
<td>ee$appell</td>
<td>ee</td>
<td>ee$</td>
<td>$ap</td>
</tr>
<tr>
<td>6</td>
<td>elle$app</td>
<td>el</td>
<td>el</td>
<td>$a</td>
</tr>
<tr>
<td>7</td>
<td>lee$appel</td>
<td>le</td>
<td>le</td>
<td>ap</td>
</tr>
<tr>
<td>8</td>
<td>llee$app</td>
<td>ll</td>
<td>ll</td>
<td>$a</td>
</tr>
<tr>
<td>9</td>
<td>pelle$ape</td>
<td>pe</td>
<td>pe</td>
<td>a</td>
</tr>
<tr>
<td>10</td>
<td>ppelle$a</td>
<td>pp</td>
<td>pp</td>
<td>a</td>
</tr>
<tr>
<td>11</td>
<td>p$pelle</td>
<td>pp</td>
<td>pp</td>
<td>a</td>
</tr>
<tr>
<td>12</td>
<td>ppe$pelle</td>
<td>pp</td>
<td>pp</td>
<td>a</td>
</tr>
<tr>
<td>13</td>
<td>app$pelle</td>
<td>pp</td>
<td>pp</td>
<td>a</td>
</tr>
<tr>
<td>14</td>
<td>ppe$pelle</td>
<td>pp</td>
<td>pp</td>
<td>a</td>
</tr>
</tbody>
</table>

**BWT sort**: Sort the BWT of the string. 
**prepend BWT column**: Prepend the BWT column to the sorted columns. 
**sort these 2 columns**: Sort the first 2 columns. 
**sort these 3 columns**: Sort the first 3 columns.
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

| dogwood$ | $dogwood | ogwood$d | d$dogwoo | gwood$do | dogwood$ | wood$dog | dogwood | ood$dogw | od$dogwo | ogwood$d | od$dogwoo | ood$dogw | dogwood | wood$dog |
|----------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|

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

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>$</td>
<td>d</td>
<td>$</td>
<td>d</td>
<td>$</td>
</tr>
<tr>
<td>o</td>
<td>d</td>
<td>$</td>
<td>d</td>
<td>$</td>
<td>d</td>
</tr>
<tr>
<td>$</td>
<td>d</td>
<td>$</td>
<td>d</td>
<td>d</td>
<td>$</td>
</tr>
<tr>
<td>o</td>
<td>g</td>
<td>g</td>
<td>w</td>
<td>g</td>
<td>w</td>
</tr>
<tr>
<td>o</td>
<td>o</td>
<td>o</td>
<td>g</td>
<td>w</td>
<td>w</td>
</tr>
<tr>
<td>g</td>
<td>w</td>
<td>w</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>d</td>
<td>o</td>
<td>o</td>
<td>g</td>
<td>w</td>
<td>w</td>
</tr>
<tr>
<td>w</td>
<td>o</td>
<td>o</td>
<td>g</td>
<td>w</td>
<td>w</td>
</tr>
</tbody>
</table>

Prepend  Sort  Prepend  Sort  Prepend  Sort  Prepend  Sort
Searching with BWT: Occ Mapping

**Occ Mapping**

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

# of times letter appears before this position in the last column.

**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.
Searching with BWT: Occ Mapping

Occ Mapping

\[
\text{BWT}(\text{unabashable})
\]

$\sum$

\[
\begin{array}{cccccccccc}
\$ \ a \ b \ e \ h \ l \ n \ s \ u \$
\end{array}
\]

# of times letter appears before this position in the last column.

\[
\sum \text{Occ}(\text{BWT}(S)[i], i)
\]

\[
\sum \text{Occ}(\text{BWT}(S)[i], i)
\]

\[
\text{LF}(i) = C[\text{BWT}(S)[i]] + \text{Occ}(\text{BWT}(S)[i], i)
\]

The character in row $i$ of the last column of the BWT matrix is the same as the character in row $\text{LF}(i)$ of the first column of the BWT matrix.

\[
\text{LF Property: The } i^{th} \text{ occurrence of a letter } X \text{ in the last column corresponds to the } i^{th} \text{ occurrence of } X \text{ in the first column.}
\]
**BWT Search**

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

---

**Occ Mapping**

<table>
<thead>
<tr>
<th></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>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>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</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>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<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>
<td>0</td>
</tr>
<tr>
<td>0</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>0</td>
<td>1</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>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>
<td>0</td>
</tr>
<tr>
<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>
<td>0</td>
</tr>
<tr>
<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>
<td>0</td>
</tr>
<tr>
<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>
<td>0</td>
</tr>
<tr>
<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>
<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>1</td>
</tr>
</tbody>
</table>

---

**Step 1:** Find the range of “a”s in the first column

**Step 2:** Look at the same range in the last column.

**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.

**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.
BWT Searching Example 2

pattern = “bana”

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

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

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

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

• Don’t have to store the Occ 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 $C[c] = \#$ of occurrences in the string of characters lexicographically $< c$.
  • Then start of the “$a$” range, for example, is: $C[“a”] + 1$.

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

$s = \text{appellee}\$

1 2 3 4 5 6 7 8 9

$\text{appellee}$

appellee$

e$appellee

ee$appellee

elee$appellee

ellee$appellee

BWT matrix

The suffixes are obtained by deleting everything after the $ $

Suffix array (start position for the suffixes)

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

($$ is a special case)

$\text{appellee}$

appellee$

e$appellee

ee$appellee

elee$appellee

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

subtract 1

9
1
8
7
4
6
5
3
2

$s[9-1] = e$

$s[1-1] = $$

$s[8-1] = e$

$s[7-1] = l$

$s[4-1] = p$

$s[6-1] = l$

$s[5-1] = e$

$s[3-1] = p$

$s[2-1] = a$
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:

$\Sigma = \{\$, e, l, p\}$
$s = appellee$  
$\ 123456789$
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. (Bowtie e.g. uses [1])

• 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.

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:

replace runs of 0s with the **count** of 0s

\[
\text{PrefixCode}(\text{rle}(\text{MTF}(\text{BWT}(S)))))
\]

Huffman code that uses more bits for rare symbols
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.

$$
\sum \ 
\begin{array}{c|c}
\text{do$oodwg} & 13210244 \\
\$dgow & 1 \\
d$gow & 13 \\
o$dgw & 132 \\
$odgw & 1321 \\
o$dgw & 13210 \\
d$gow & 132102 \\
wo$g & 1321024 \\
\end{array}
$$

List with all letters from the allowed alphabet

Benefits:

- Runs of the same letter will lead to runs of 0s.
- Common letters get small numbers, while rare letters get big numbers.
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] + \text{Occ}(c, sp-1) + 1
        ep = C[c] + \text{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.

\text{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):

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

\( u/L \) blocks, each array is \(|\Sigma|\log L\) space
\[ \Rightarrow \frac{u}{L} \log L = \frac{u}{\log u} \log \log u \] 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**

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

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

\( \frac{u}{L} \) blocks, each array is \( |\Sigma| \log L \) space
\[ \Rightarrow \frac{u}{L} \log L = \frac{u}{\log u} \log \log u \text{ total space.} \]

superblock: store \( |\Sigma| \)-long array giving \# of occurrences of each character up thru and including this superblock

\( \frac{u}{L^2} \) superblocks, each array is \( |\Sigma| \log u \) long 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.
Extra Info to Compute Occ

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

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

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

\[ \text{Occ}(c, p) = \text{\# of “c” up thru } p: \]
sum value at last superblock, value at end of previous block, but then need to handle \textit{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: \( O(|\Sigma|L2^L|\Sigma|) = O(L2^L) = O(u^c \log u) \) for \( c < 1 \) (since the string is compressed)
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.