Burrows-Wheeler Transform

CMSC 423
Motivation - Short Read Mapping

A Cow Genome

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: http://genomebiology.com/2009/10/3/R25

**BWA**

*Fast and accurate short read alignment with Burrows–Wheeler transform*

Heng Li and Richard Durbin

Author Affiliations

* To whom correspondence should be addressed.

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

### Bowtie Performance

<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

BWT(banana) = 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

BWT sort BWT → first column

$e$
a
$a$
app
appelle
elle$ap$
lee$apel$
elle$appe$
pelle$ap$
pelle$ape$
ap$pp$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$
pee$

def inverseBWT(s):
    B = [s₁, s₂, s₃, ..., sₙ]
    for i = 1..n:
        sort B
        prepend sᵢ to B[i]
    return row of B that ends with $
Another BWT Example

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

`BWT(dogwood$) = do$ooodwg`
Another BWT Example

```
<table>
<thead>
<tr>
<th></th>
<th>Prepend</th>
<th>Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>$d</td>
<td>d</td>
</tr>
<tr>
<td>o</td>
<td>d$</td>
<td>$d</td>
</tr>
<tr>
<td>$</td>
<td>d</td>
<td>do</td>
</tr>
<tr>
<td>o</td>
<td>g</td>
<td>gw</td>
</tr>
<tr>
<td>o</td>
<td>o</td>
<td>od</td>
</tr>
<tr>
<td>d</td>
<td>o</td>
<td>og</td>
</tr>
<tr>
<td>w</td>
<td>o</td>
<td>wo</td>
</tr>
<tr>
<td>g</td>
<td>w</td>
<td>wo</td>
</tr>
<tr>
<td></td>
<td>$dog</td>
<td>dogw</td>
</tr>
<tr>
<td></td>
<td>d$dog</td>
<td>dogw</td>
</tr>
<tr>
<td></td>
<td>dogwo</td>
<td>dogw</td>
</tr>
<tr>
<td></td>
<td>gwood</td>
<td>gwood</td>
</tr>
<tr>
<td></td>
<td>od$d</td>
<td>do$</td>
</tr>
<tr>
<td></td>
<td>d$dog</td>
<td>dogw</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>Prepend</th>
<th>Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>$dogw</td>
<td>$d</td>
</tr>
<tr>
<td>o</td>
<td>d$dog</td>
<td>d$</td>
</tr>
<tr>
<td>$</td>
<td>dogwo</td>
<td>dog</td>
</tr>
<tr>
<td>o</td>
<td>ood$</td>
<td>ood</td>
</tr>
<tr>
<td>d</td>
<td>woodd</td>
<td>wood</td>
</tr>
<tr>
<td>w</td>
<td>ood$</td>
<td>ood</td>
</tr>
<tr>
<td>g</td>
<td>wood$</td>
<td>wood</td>
</tr>
<tr>
<td></td>
<td>dogwoo</td>
<td>dog</td>
</tr>
<tr>
<td></td>
<td>ood$</td>
<td>dog</td>
</tr>
<tr>
<td></td>
<td>gwood</td>
<td>wood</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>Prepend</th>
<th>Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>dogwo</td>
<td>dog</td>
</tr>
<tr>
<td>o</td>
<td>d$dog</td>
<td>dog</td>
</tr>
<tr>
<td>$</td>
<td>dogwo</td>
<td>dog</td>
</tr>
<tr>
<td>o</td>
<td>ood$</td>
<td>ood</td>
</tr>
<tr>
<td>d</td>
<td>woodd</td>
<td>wood</td>
</tr>
<tr>
<td>w</td>
<td>ood$</td>
<td>ood</td>
</tr>
<tr>
<td>g</td>
<td>wood$</td>
<td>wood</td>
</tr>
<tr>
<td></td>
<td>dogwoo</td>
<td>dog</td>
</tr>
<tr>
<td></td>
<td>ood$</td>
<td>dog</td>
</tr>
<tr>
<td></td>
<td>gwood</td>
<td>wood</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>Prepend</th>
<th>Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>dogwoo</td>
<td>dog</td>
</tr>
<tr>
<td>o</td>
<td>d$dog</td>
<td>dog</td>
</tr>
<tr>
<td>$</td>
<td>dogwo</td>
<td>dog</td>
</tr>
<tr>
<td>o</td>
<td>ood$</td>
<td>ood</td>
</tr>
<tr>
<td>d</td>
<td>woodd</td>
<td>wood</td>
</tr>
<tr>
<td>w</td>
<td>ood$</td>
<td>ood</td>
</tr>
<tr>
<td>g</td>
<td>wood$</td>
<td>wood</td>
</tr>
<tr>
<td></td>
<td>dogwoo</td>
<td>dog</td>
</tr>
<tr>
<td></td>
<td>ood$</td>
<td>dog</td>
</tr>
<tr>
<td></td>
<td>gwood</td>
<td>wood</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>Prepend</th>
<th>Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>dogwood</td>
<td>dog</td>
</tr>
<tr>
<td>o</td>
<td>d$dog</td>
<td>dog</td>
</tr>
<tr>
<td>$</td>
<td>dogwo</td>
<td>dog</td>
</tr>
<tr>
<td>o</td>
<td>ood$</td>
<td>ood</td>
</tr>
<tr>
<td>d</td>
<td>woodd</td>
<td>wood</td>
</tr>
<tr>
<td>w</td>
<td>ood$</td>
<td>ood</td>
</tr>
<tr>
<td>g</td>
<td>wood$</td>
<td>wood</td>
</tr>
<tr>
<td></td>
<td>dogwoo</td>
<td>dog</td>
</tr>
<tr>
<td></td>
<td>ood$</td>
<td>dog</td>
</tr>
<tr>
<td></td>
<td>gwood</td>
<td>wood</td>
</tr>
</tbody>
</table>
```
Searching with BWT: LF Mapping

<table>
<thead>
<tr>
<th>BWT(unabashable)</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>
</tr>
<tr>
<td>abashable$un</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>able$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>
</tr>
<tr>
<td>ashable$unab</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>bashable$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>
</tr>
<tr>
<td>ble$unabasha</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>e$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>
</tr>
<tr>
<td>hable$unabas</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>le$unabashbab</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>
</tr>
<tr>
<td>nabashable$u</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>shable$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>
</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>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$\sum$ # 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.
BWT Search

BWTS\text{earch}(aba)  \quad \text{Start from the } \textbf{end} \text{ of the pattern}

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”

(B,E) = 0, 2

(B,E) = 1, 2

(B,E) = 0, 1
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 \textit{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: \(O(|\text{pattern}|)\)
  • 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

<table>
<thead>
<tr>
<th>BWT matrix</th>
<th>Suffix array (start position for the suffixes)</th>
<th>Suffix position - 1 = the position of the last character of the BWT matrix ($ is a special case)</th>
</tr>
</thead>
</table>
| $appellee$ | $\begin{array}{c}9 \\ 1 \\ 8 \\ 7 \\ 4 \\ 6 \\ 5 \\ 3 \\ 2 \end{array}$ | $s[9-1] = e$
| appellee$ |                                           | $s[1-1] = $ |
| e$appelle |                                           | $s[8-1] = e$ |
| ee$appell |                                           | $s[7-1] = l$ |
| ellee$app  |                                           | $s[4-1] = p$ |
| lee$appel |                                           | $s[6-1] = l$ |
| llee$appe |                                           | $s[5-1] = e$ |
| pellee$ap |                                           | $s[3-1] = p$ |
| ppellee$a |                                           | $s[2-1] = a$ |
| $appellee$ |                                           | |
| appellee$  |                                           | |
| e$appelle  |                                           | |
| ee$appell  |                                           | |
| ellee$app  |                                           | |
| lee$appel  |                                           | |
| llee$appe  |                                           | |
| pellee$ap  |                                           | |
| ppellee$a  |                                           | |

The suffixes are obtained by deleting everything after the $.

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.
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.
# 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.

<table>
<thead>
<tr>
<th>List with all letters from the allowed alphabet</th>
<th>do$oodwg</th>
</tr>
</thead>
<tbody>
<tr>
<td>$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>

\[ \sum \]

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:

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

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

\[\text{function Count}(S_{bwt}, P):\]

\[c = P[p], i = p\]

\[sp = C[c] + 1; ep = C[c+1]\]

\[\text{while } (sp \leq ep) \text{ and } (i \geq 2) \text{ do}\]

\[c = P[i-1]\]

\[sp = C[c] + \text{Occ}(c, sp-1) + 1\]

\[ep = C[c] + \text{Occ}(c, ep)\]

\[i = i - 1\]

\[\text{if } ep < sp \text{ then}\]

\[\text{return } \text{"not found"}\]

\[\text{else}\]

\[\text{return } ep - sp + 1\]

\[C[c] = \text{index into first column where the "c"s begin.}\]

\[\text{Occ}(c, p) = \text{# of } c \text{ in the first } p \text{ characters of BWT}(S), \text{aka the LF mapping.}\]
Computing Occ in Compressed String

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

\[
\text{BWT}(S) = \begin{array}{ccccccc}
BT_1 & BT_2 & BT_3 & \ldots & & & \\
\end{array}
\]

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 \(\text{Occ}(c, p)\) quickly.

\[
\text{Occ}(c, p) = \# \text{ of “c” up thru } p
\]
**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 \( L = O(\log u) \)

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

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

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

\[ \text{u/L}\text{ superblocks, each array is } |\Sigma|\log u \text{ long } \Rightarrow \]
\[ \frac{u}{(\log u)^2} \log u = \frac{u}{\log u} \text{ total space.} \]
**Extra Info to Compute Occ**

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

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

\[ \text{Occ}(c, p) = \text{# of \textit{"c"} up thru } p: \]
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: \( O(|\Sigma|L^2|\Sigma|) = O(L^2L') = O(u^c\log u) \) 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.