Burrows-Wheeler-Transform

• Create all cyclic permutations of the input string.
• Sort them lexicographically.
• Output the last character of each permutation, and the position of the original text in the sorted table.
Burrows-Wheeler-Transform

- **Source text:** cacbcaabca

- **Output:** caccccabbaa, 8
Reversing the Burrows-Wheeler-Transform

• Transformed input: cacccabbaa, 8
• Table: left side: sorted, right side: original
  a(1) -> c(1) 0
  a(2) -> a(1) 1
  a(3) -> c(2) 2
  a(4) -> c(3) 3
  b(1) -> c(4) 4
  b(2) -> a(2) 5
  c(1) -> b(1) 6
  c(2) -> b(2) 7
  c(3) -> a(3) 8 <-
  c(4) -> a(4) 9

Output: cacbcaabca ←
BWT back-end

- Move-to-front (MTF) encoding
  - Index: 012
- Input: cacccabbbaa, 8, Alphabet: abc
  - 2, cab
  - 1, acb
  - 1, cab
  - 0, cab
  - 0, cab
  - 1, acb
  - 2, bac
  - 0, bac
  - 1, abc
  - 0, abc
- Output: 2110012010, 8
BWT back-end

• Huffman coding or arithmetic coding.
• Input: 2110012010, 8
• Encode MTF output string, add position (‘8’) to output in binary.
Burrows-Wheeler Transform

- **Forward Transform**: bananas

<table>
<thead>
<tr>
<th>M’</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>bananas</td>
<td>ananasb</td>
</tr>
<tr>
<td>sbanana</td>
<td>anasban</td>
</tr>
<tr>
<td>asbanan</td>
<td>asbanan</td>
</tr>
<tr>
<td>nasbana</td>
<td>bananas</td>
</tr>
<tr>
<td>anasban</td>
<td>nanasba</td>
</tr>
<tr>
<td>nanasba</td>
<td>nasbana</td>
</tr>
<tr>
<td>ananasb</td>
<td>sbanana</td>
</tr>
</tbody>
</table>

(Stable Sort: preserves original order of records with equal keys.)

M contains all the suffixes of the text and they are sorted!
Burrows-Wheeler Transform

• Forward Transform

Given an input text \( T = t_1 t_2 \ldots t_u \)

1. Form \( u \) permutations of \( T \) by cyclic rotations of characters in \( T \), forming a \( u \times u \) matrix \( M' \), with each row representing one permutation of \( T \).

2. Sort the rows of \( M' \) lexicographically to form another matrix \( M \), including \( T \) as one of rows.

3. Record \( L \), the last column of \( M \), and \( id \), the row number where \( T \) is in \( M \).

4. Output of BWT is \( (L, id) \).
Burrows-Wheeler Transform

• **Inverse Transform:**
  Given only the \((L, id)\) pair
  1. (Stable) Sort \(L\) to produce \(F\), the array of first characters
  2. Compute the index array \(V\). \(V\) provides 1-1 mapping between the elements of \(L\) and \(F\).
  3. \(F[V[j]] = L[j]\). That is \(V(j) = i\) if \(F(i) = L(j)\).
  4. Generate original text \(T\). The symbol \(L[j]\) cyclically precedes the symbol \(F[j]\) in \(T\), that is, \(L[V[j]]\) cyclically precedes the symbol \(L[j]\) in \(T\) except when \(j\) equals \(id\) and \(L(id) = T(n)\), \(n\) is the length of \(T\).

Given \(V\) and \(L\), the algorithm to generate \(T\) can be written as:

\[
T[n+1-i] = L[V[id]], \quad i=1,2,...,n \\
\text{where} \quad V^1[s] = s
\]

and

\[
V^{i+1}[s] = V[V^i[s]], \quad 1 \leq s \leq n.
\]
Burrows-Wheeler Transform

\[
\begin{array}{c}
F \\
a \quad a \quad a \\
a \quad n \\
b \\
n \\
n \\
s \\
L \\
b \\
n \\
 n \\
s \\
 a \\
 a \\
 a \\
 id = 4
\end{array}
\]
BWT Based Compression

- BWT
- Move-to-Front (*MTF*)
- Run-Length-Encoding (RLE)
- Variance Length Encoding (VLE)
  - Huffman or
  - Arithmetic

Input -> **BWT** -> **MTF** -> **RLE** -> **VLE** -> Output
Derivation of Auxiliary Arrays

- T=abraca:

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>...</th>
<th>L</th>
<th>V</th>
<th>FS</th>
<th>FST</th>
<th>......</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>c</td>
<td>5</td>
<td>aa</td>
<td>aab</td>
<td>a</td>
<td>ab</td>
<td>abraca</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>1</td>
<td>ab</td>
<td>abr</td>
<td>a</td>
<td>ab</td>
<td>abrac</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>r</td>
<td>6</td>
<td>ac</td>
<td>aca</td>
<td>a</td>
<td>ac</td>
<td>acaabr</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>a</td>
<td>2</td>
<td>br</td>
<td>bra</td>
<td>b</td>
<td>br</td>
<td>braca</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>a</td>
<td>3</td>
<td>ca</td>
<td>caa</td>
<td>c</td>
<td>ca</td>
<td>caabra</td>
</tr>
<tr>
<td>r</td>
<td>r</td>
<td>b</td>
<td>4</td>
<td>ra</td>
<td>rac</td>
<td>r</td>
<td>ra</td>
<td>racaab</td>
</tr>
</tbody>
</table>
Derivation of Auxiliary Arrays

- Generating q-grams from BWT output

  Grams(F,L,V,q)
  
  F(1-gram) = F;
  
  for x = 2 to q do
    for i = 1 to u do
      F(x-gram)[V(i)] := L[i]*F((x-1)-gram)[i];
    end;
  end;

* Denotes concatenation. If q=2, the result will be the sorted bi-grams;
BZip2

- Run-length encoding
- Burrows-Wheeler-Transform
- Modified Move-To-Front-Encoding
- Huffman encoding (older versions: arithmetic coding).
PPM (Partial Predicate Match) and BWT (Burrows-Wheeler Transform) Methods

- Adaptive finite-context models
- Zero-Frequency problem
- Escape probabilities, Exclusion
- PPMC, PPMD, PPMD+, PPM*

- Forward context
- Uses run-length and Move-To-Front Coding followed by Huffman, bzip2
- BWT suffix trees
<table>
<thead>
<tr>
<th>Order k=2</th>
<th>Order k=1</th>
<th>Order k=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ab&gt;r 2</td>
<td>2/3</td>
<td>a&gt;b 2</td>
</tr>
<tr>
<td>&gt;Esc 1</td>
<td>1/3</td>
<td>a&gt;c 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a&gt;d 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;Esc 3</td>
</tr>
<tr>
<td>ac&gt;a 1</td>
<td>1/2</td>
<td>r&gt;a 2</td>
</tr>
<tr>
<td>&gt;Esc 1</td>
<td>1/2</td>
<td>Esc 1</td>
</tr>
<tr>
<td>b&gt;r 2</td>
<td>2/3</td>
<td>&gt;Esc 5</td>
</tr>
<tr>
<td>ad&gt;a 1</td>
<td>1/2</td>
<td>&gt;Esc 1</td>
</tr>
<tr>
<td>&gt;Esc 1</td>
<td>1/2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>c&gt;a 1</td>
</tr>
<tr>
<td>br&gt;a 2</td>
<td>2/3</td>
<td>&gt;Esc 1</td>
</tr>
<tr>
<td>&gt;Esc 1</td>
<td>1/3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>d&gt;a 1</td>
</tr>
<tr>
<td>ca&gt;d 1</td>
<td>1/2</td>
<td>Esc 1</td>
</tr>
<tr>
<td>&gt;Esc 1</td>
<td>1/2, da&gt;b 1 1/2, Esc 1 1/2</td>
<td>ra&gt;c 1 1/2, Esc 1 1/2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Char</th>
<th>Probabilities Without Exclusion</th>
<th>Probabilities With Exclusion</th>
<th>No. of bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>½</td>
<td>½</td>
<td>-log(1/2)=1 bit</td>
</tr>
<tr>
<td>d</td>
<td>½, 1/7</td>
<td>½, 1/6</td>
<td>-log(1/2*1/6)=3.6 bits</td>
</tr>
<tr>
<td>t</td>
<td>½,3/7,5/16,1/</td>
<td>A</td>
<td>½,3/6,5/12,1/(</td>
</tr>
</tbody>
</table>

Note for ‘d’, for Order(1) model the probability is increased to 1/6 (rather than 1/7) with exclusion. This is because ‘c’ appeared in the context of ‘ra’ and therefore the context a->c will never be used at lower level and therefore should be excluded in estimating the probabilities at level 1. Similarly, the escape probability for t at Order (1) is reduced to 3/6 and so on. Since ‘b’, ‘c’ and ‘d’ appear in the context of ‘a->’, these are removed from Order(0) context, so that Esc probability for Order(0) is 5/12. In order (-1), the Esc probability is 1/251 since 5 characters appear before (256-5=251). Note, Sayood describes a slightly different method for Esc probability estimate.
PPMC

- Multiple levels, user-adjustable
- Escape probabilities calculates using Method C.
- Exclusion
Main differences

• PPM predicts the following character. BWT predicts the preceding character.
• BWT does not explicitly use the context (except in sorting algorithms), PPM does.
• PPM limits context depth. BWT does not. Not limiting context depth can be a disadvantage.
In PPM* the same unique context is used to predict the next characters (L).