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**
- aabcacacb**c** 0
  abcacacbc**a** 1
  acacbcaab**c** 2
  acbcaabca**c** 3
  bcaabcaca**c** 4
  bcacacbca**a** 5
  caabcacac**b** 6
  cacacbcaa**b** 7
  cacbcaabc**a** **8** <- original
  cbcaabcac**a** 9

- Output: **cacccabbaa, 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: **`caccccabbaa, 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

M'		M		
bananas		ananasb		
sbanana		anasban		
asbanan		asbanan		
nasbana	→	**bananas**	→	bnnsaaa,4
anasban		nanasba		
nanasba		nasbana		
ananasb		sbanana		

(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*x*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*.:

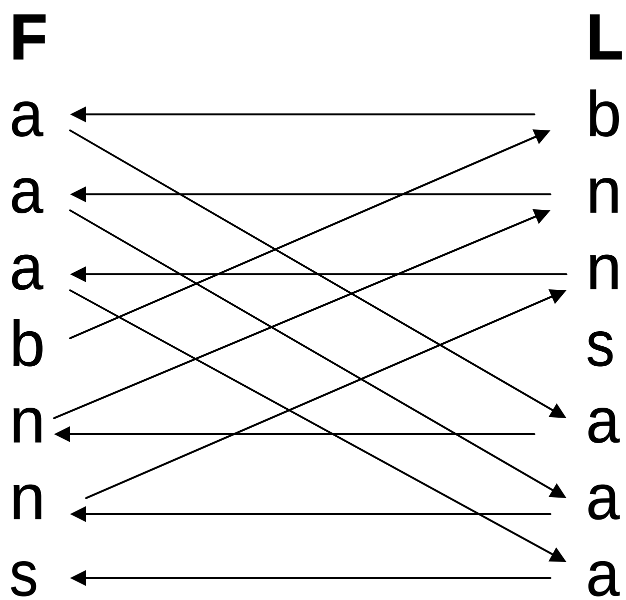
Given *V* and *L*, the algorithm to generate *T* can be written as:

$T[n+1-i] = L[V^i[id]],\ i = 1,2,\ldots,n.$ where $V^1[s] = s$

and

$$V^{i+1}[s] = V[V^i[s]],\ 1 \le s \le n.$$

Burrows-Wheeler Transform



id =4

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:

F ….	L	V	FS	FST ……	M
a	c	5	aa	aab	aabrac
a	a	1	ab	abr	abraca
a	r	6	ac	aca	acaabr
b	a	2	br	bra	bracaa
c	a	3	ca	caa	caabra
r	b	4	ra	rac	racaab

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>
<tr><th colspan="3">Order k=2</th><th colspan="3">Order k=1</th><th colspan="3">Order k=0</th></tr>
<tr><th>Pr.</th><th>Cn.</th><th>P</th><th>Pr.</th><th>Cn.</th><th>P</th><th>Pr.</th><th>Cn.</th><th>P</th></tr>
<tr><td colspan="9">***</td></tr>
<tr><td>ab>r</td><td>2</td><td>2/3</td><td>a>b</td><td>2</td><td>2/7</td><td>>a</td><td>5</td><td>5/16</td></tr>
<tr><td>>Esc</td><td>1</td><td>1/3</td><td>a>c</td><td>1</td><td>1/7</td><td>>b</td><td>2</td><td>2/16</td></tr>
<tr><td></td><td></td><td></td><td>a>d</td><td>1</td><td>1/7</td><td>>c</td><td>1</td><td>1/16</td></tr>
<tr><td></td><td></td><td></td><td>>Esc</td><td>3</td><td>3/7</td><td>>d</td><td>1</td><td>1/16</td></tr>
<tr><td>ac>a</td><td>1</td><td>½</td><td>r>a</td><td>2</td><td>2/3</td><td>>r</td><td>2</td><td>2/16</td></tr>
<tr><td>>Esc</td><td>1</td><td>½</td><td>Esc</td><td>1</td><td>1/3</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td>b>r</td><td>2</td><td>2/3</td><td>>Esc</td><td>5</td><td>5/16</td></tr>
<tr><td>ad>a</td><td>1</td><td>½</td><td>>Esc</td><td>1</td><td>1/3</td><td></td><td></td><td></td></tr>
<tr><td>>Esc</td><td>1</td><td>½</td><td></td><td></td><td></td><td colspan="3">-----------------</td></tr>
<tr><td></td><td></td><td></td><td>c>a</td><td>1</td><td>½</td><td colspan="3">Order k=-1</td></tr>
<tr><td>br>a</td><td>2</td><td>2/3</td><td>>Esc</td><td>1</td><td>½</td><td>Pr.</td><td>Cn.</td><td>P</td></tr>
<tr><td>>Esc</td><td>1</td><td>1/3</td><td></td><td></td><td></td><td colspan="3">-----------------</td></tr>
<tr><td></td><td></td><td></td><td>d>a</td><td>1</td><td>½</td><td>> A</td><td>1</td><td>1/|A|</td></tr>
<tr><td>ca>d</td><td>1</td><td>½</td><td>Esc</td><td>1</td><td>½</td><td colspan="3">-----------------</td></tr>
</table>

>Esc 1 ½ , da>b 1 ½, Esc 1 ½ **ra>c 1 ½, Esc 1 ½**

PPMC model for the string ‘**abracadabra**’(Cleary-Teahan,Computer Journal,Vol.36,No.5,1993)

Char	Probabilities Without Exclusion	Probabilities With Exclusion	No. of bits
c	½	½	-log(1/2)=1 bit
d	½, 1/7	½, 1/6	-log(1/2*1/6)=3.6 bits
t	½,3/7,5/16,1/|A|	½,3/6,5/12,1/(|A|-5)	-log(1/2.3/6.5/12.1/251)=11.2 bits(?)

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 descibes 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.

Matrix	Sorted Matrix	Unique Context following L	L
abracadabra#	**#**abracadabr**a**	#	a
#abracadabra	**a#**abracadab**r**	a#	r
a#abracadabr	**abra#**abraca**d**	abra#	d
ra#abracadab	**abrac**adabra**#** (4)	abrac	#
bra#abracada	**ac**adabra#ab**r**	ac	r
abra#abracad	**ad**abra#abra**c**	ad	c
dabra#abraca	**bra#**abracad**a**	bra#	a
adabra#abrac	**brac**adabra#**a**	brac	a
cadabra#abra	**c**adabra#abr**a**	c	a
acadabra#abr	**d**abra#abrac**a**	d	a
racadabra#ab	**ra#**abracada**b**	ra#	b
bracadabra#a	**rac**adabra#a**b**	rac	b

In PPM* the same unique context is used to predict the next characters(L)