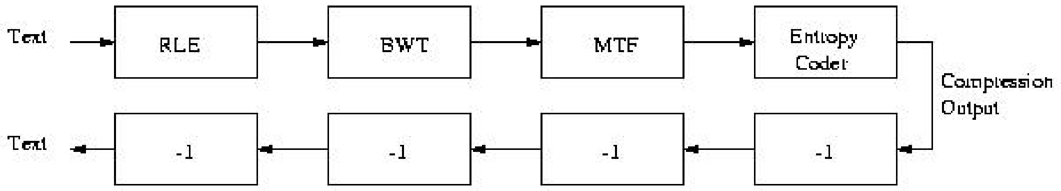
Burrows Wheeler Transform (BWT)

- BWT was invented in 1983 in Cambridge by Wheeler.
- Lossless reversible transformation (not a compression algorithm) applicable to sequence of symbols given as a block – like a block in DCT – but extension to 2 dimensional image has not yet been done. It is a hot research topic.
- Compression system: (the bottom line of boxes denote inverse operation of the corresponding box in the upper line)



- Not an on-line or streaming model algorithm. The entire block must be available to the transform. Typically the block size is 100kb to 900kb.
- Informal description:

  Example 1: *T = abraca*

```
          M'                      M
                               F    L
       abraca                  aabrac
       aabrac                  abraca    <- id
       caabra                  acaabr
       acaabr                  bracaa
       racaab                  caabra
       bracaa                  racaab
Output: (caraab,2)
```

  Note: text is decoded backwards.

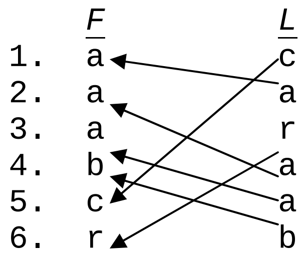
- *L* with *id* is actually an expansion than compression. But it shows a pattern of concentration of a few symbols in any region. Note, except for the row *id*, the last column symbols precede the corresponding first column symbols in the original text. Thus, the symbols that have same forward context tend to cluster together in column *L*. This will be better illustrated with an example below:

  T = swiss-miss

```
    1.  –missswiss
    2.  iss-missssw
    3.  issswiss-m
    4.  missswiss-
    5.  s-missswis
    6.  ss-missswi
    7.  ssswiss-mi
    8.  sswiss-mis
    9.  swiss-miss               Output:(swm-siisss,9)
    10. wiss-misss
```

Stated in other words, if a symbol ‘s’ appears at certain position in *L*, then other occurrences of ‘s’ are likely to be found nearby. This property means that *L* can be further decorrelated by Move-To-Front (MTF) algorithm.

- Inverse transform: Given *L*, we can obtain *F* by sorting *L* in linear time.



id=2

abraca

Two properties have been utilized:
1) Stable sort.
2) *L(j)* precede *F(j)* in *T*, *except when j equals id* and *L(id) = T(n), n* is the length of *T*.

Define an index vector *V* providing a one-to-one mapping between the elements of *L* and *F,* the pair of indices corresponding to back arrows in the above diagram..

V = (5,1,6,2,3,4)

That is *V(j) = i* if *F(i) = L(j)* or *F(V(j)) = L(j)*.
Given *V* and *L*, the algorithm to generate *T* can be written as:

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

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

```
u=6                 123456
                  T=abraca
```

i=1 $T[6] = L[V^1(id)] = L(id) = a$

i=2 $T[5] = L[V^2(id)] = L(V(id)) = L(1) = c$

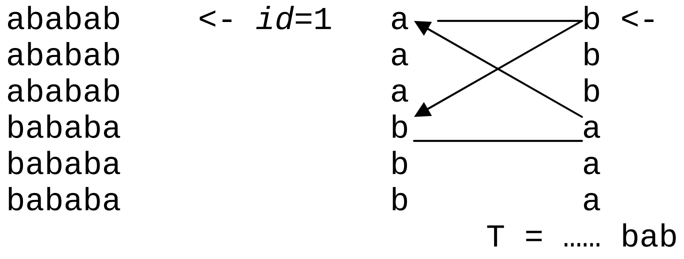
i=3 $T[4] = L[V^3(id)] = L(V(1)) = L(5) = a$

i=4 $T[3] = L[V^4(id)] = L(V(5)) = L(3) = r$

i=5 $T[2] = L[V^5(id)] = L(V(3)) = L(6) = b$

i=6 $T[1] = L[V^6(id)] = L(V(6)) = L(4) = a$

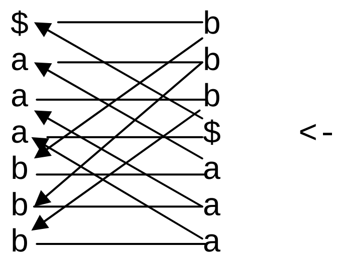
Now, let’s take *T* = *ababab*



The same thing will happen if T = "aaaa". Unless we know the length of T, the algorithm will go into a loop. The problem can be resolved by putting an eof symbol '\$' sign, considered smallest.

T=ababab\$

```
    $ababab
    ab$abab
    abab$ab
    ababab$   <- id=4
    b$ababa
    bab$aba
    babab$a
```



As soon as it enters '\$' back, it stops. Note the '\$' delineate all the proper suffixes of T in lexicographic sorted order. Thus we can see now how BWT captures all forward context of arbitrary length in sorted order. This is of extreme significance or we will see later when we discuss PPM.

T = abraca\$

	V	Sorted suffix
1. \$abraca	2	\$
2. a\$abrac	6	a\$
3. abraca\$	1	abraca\$
4. aca\$abr	7	aca\$
5. braca\$a	3	braca\$
6. ca\$abra	4	ca\$
7. raca\$ab	5	raca\$

V = (2,6,1,7,3,4,5)

In practice, V is computed using two arrays.

Character count array $C = (c_1, c_2, ..., c_{|\Sigma|})$. $C(c)$ = the number of occurrence in L of all the character preceding σ_c, the c-th symbol in $\Sigma = (\$, a, b, c, r)$. Let

R(j) = the number of occurrence of *L(j)* in the prefix *L(1,2,…j)* of *L*.

While computing R, we can also compute total count array *total_count* for all the characters in *L*. C*(c)* can be obtained from *total_count* via cumulative count variable *cum_count* with *O(|Σ|)* time.

L=M=ac\$raab, M stands for the message that the decoder receives.

Σ	*total_count* in M	*cum_count*	$C(c)$ = *cum_count* - 1
\$	1	1	0
a	3	4	1
b	1	5	4
c	1	6	5

r	1	7	6

It is also possible to compute *C(c)* as *C(c) = (position(c) -1)*, where *position(c)* is the index in *F* where the group of characters σ_c start occurring. Thus,

$$V(i)= C(L(i))+ R(i), \qquad i=1,..,n$$

For our example *L=M=ac$raab*, $\Sigma = (\$, a, b, c, r)$,*C*= (0,1,4,5,6), R=(1, 1,1,1,1,2,3,1), we get *V=(2,6,1,7,3,4,5)*

	F	*L*	*R*=cum occurrence in M in the same group
1	$	a	1
2	a	c	1
3	a	$	1
4	a	r	1
5	b	a	2
6	c	a	3
7	r	b	1

Given *L=(ac$raab)*, we can obtain *R* and *C* vectors as follows:

Given $\Sigma = (1,2,3,...|\Sigma|= s)$ /* alphabet is renamed as integers */

```
    for  i = 1 to s  do  total_count(i)=0;
    for  j = 1 to n  do
         // total_count contains the cumulative count until L(j)
         total_count(L(j)) = total_count(L(j))+1;
         R(j)=total_count(L(j));
    end

    cum_count = 0;
    for i = 1 to s do
         C(i) = cum_count;
         cum_count = cum_count + total_count(i);
    end
```

$\Sigma = (1,2,3,...,5) = (\$, a, b, c, r)$ total_*count(1)=...=total_count(5)=0*

L = ac$raab

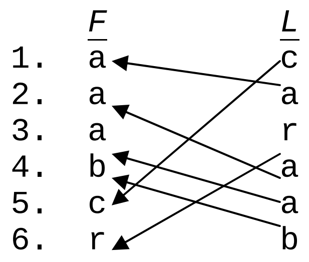
L(1) = a	*R(1) =* 1	*total_count(2)=1 /* argument denotes alphabet letter */*
L(2) = c	*R(2) = 1*	*total_count(*4*)=1*
L(3) = $	*R(3) = 1*	*total_count(1)=1*
L(4) = r	*R(4) = 1*	*total_count(5)=1*
L(5) = a	*R(5) = 2*	*total_count(2)=2*
L(6) = a	*R(6) = 3*	*total_count(2)=3*
L(7) = b	*R(7) = 1*	*total_count(3)=1*

	cum_count = 0	*total_count*
$C(1) = 0$	$cum\_count = 0{+}1 = 1$	1
$C(2) = 1$	$cum\_count = 1{+}3 = 4$	3
$C(3) = 4$	$cum\_count = 4 + 1 = 5$	1
$C(4) = 5$	$cum\_count = 5{+}1 = 6$	1
$C(5) = 6$	$cum\_count = 6{+}1 = 7$	1

$C = (0,1,4,5,6)$

Thus, given L we can obtain R and C by taking $O(n)$ time. Then we can compute V from $V(i)=R(i)+C(L(i)),\ i = 1,..,n$ by taking $O(n)$ time. Since $F(V(j))=L(j)$, we can also obtain F in $O(n)$ time from V. Thus, we can get F without sorting at the decoding side. The sorting of L would have taken $O(n \log n)$ time.

Vectors H, Hr, and Hrs.



id=2

abraca

If we follow the back arrows in sequence and list the address in column F where it lands, we get a vector call ***H=(1,5,3,6,4,2)***. These indices give the output character sequence in reverse order. Thus:

$$T = F(H(6))\ F(H(5))\ F(H(4))\ F(H(3))\ F(H(2))\ F(H(1))$$
$$= F(2)\ F(4)\ F(6)\ F(3)\ F(5)\ F(1)$$
$$\text{a} \quad \text{b} \quad \text{r} \quad \text{a} \quad \text{c} \quad \text{a}$$

Thus $F(H(i))$ gives text T in reverse order. If we reverse H to Hr, $Hr = (2,4,6,3,5,1)$ then we get the text in correct order $F(Hr(i))$

Thus we get

$$\boldsymbol{T[i] = F[Hr[i]]} \qquad \text{-(1)}$$
$$= F[H[n+1-i]]$$

How to obtain Hr from V? (Homework)

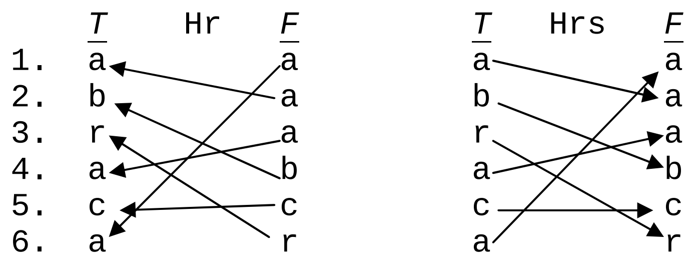
Thus we know the following:

id	T	L	F	V	H	Hr	Hrs=suffix array
1	a	c	a	5	1	2	6
2	b	a	a	1	5	4	1
3	r	r	a	6	3	6	4
4	a	a	b	2	6	3	2
5	c	a	c	3	4	5	3
6	a	b	r	4	2	1	5

Hrs is another vector that such that *Hrs(i)* gives the index of row where *i* appears in *Hr*. In other words:

T[Hrs[i]] = F[i] - (2)

Equation (1) and (2) give ont-to-one mapping between *F->T* and *T->F.*



Hrs=(6,1,4,2,5,3)

```
a
abraca
aca
braca
ca
rac
```

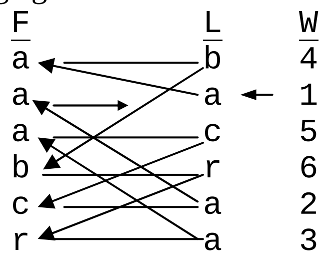
<u>Cyclic rotations</u>: In actual implementation these rotations are not done explicitly – it is done via an array of *m* pointers, one to each character of the message.

In practice, to obtain the decoded text in natural order **reverse lexicographic order** beginning from second-to-last (SL) character.

```
                         L    F
abraca                   bracaa : b
aabrac                   abraca : a    <- id=2
caabra                   caabra : c
acaabr    ->             racaab : r
racaab                   aabrac : a
bracaa                   acaabr : a
```

The column at the right hand side is the first column designated L in this reverse matrix which along with id is transmitted to the decoder.

Rename the first column *L* and the last column *F*. The row containing the original message gets the id value as before (id=2).



L(j) precedes *L(W(j))* in the text. $T = L(id)L(W(id))L(W^2(id))...L(W^{n-1}(id))$.

T = abraca