## On Burrows-Wheeler Compression

The basic principle of text compression is that, the more frequently a factor occurs, the shorter its encoding should be.

Let $c$ be a symbol and $w$ a string such that the factor $cw$ occurs frequently in the text.

- The occurrences of $cw$ may be distributed all over the text, so recognizing $cw$ as a frequently occurring factor is not easy. It requires some large, global data structures.
- In the BWT, the high frequency of $cw$ means that $c$ is frequent in that part of the BWT that corresponds to the rows of the matrix $\mathcal{M}$ beginning with $w$. This is easy to recognize using local data structures.

This localizing effect makes compressing the BWT much easier than compressing the original text.

We will not go deeper into text compression on this course.



**Example 4.23:** A part of the BWT of a reversed english text corresponding to rows beginning with `ht`:

```
oreeereoeeieeeeaooeeeeeaereeeeeeeeeeeeereeeeeeeeeeaaeeaeeeeeeee
eaeeeeeeeeaeieeeeeeeeereeeeeeeeeeeeeeeeeeeeeeeeeaeeieeeeeeaaieee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeaeieeeeeeeeeeeeeeeeeeeeeeeeeeeeaee
eeeeeeeeeeeeeeeeeereeeeeeeeeeeieaeeeeieeeeaeeeeeeeeeieeeeeeeee
eeeieeeeeeeeioaeeaoereeeeeeeeeeeaaeaaeeeeieeeeeeeieeeeeeeeaeeee
eeaeeeeeereeeaeeeeeieeeeeeeeiieee. e  eeeeiiiiii e          ,
i   o       oo e  eiiiiee,er  ,  ,     ,  . iii
```

and some of those symbols in context:

```
t raise themselves, and the hunter, thankful and r
ery night it flew round the glass mountain keeping
agon, but as soon as he threw an apple at it the b
f animals, were resting themselves.  "Halloa, comr
ple below to life.  All those who have perished on
 that the czar gave him the beautiful Princess Mil
ng of guns was heard in the distance.  The czar an
cked magician put me in this jar, sealed it with t
o acted as messenger in the golden castle flew pas
u have only to say, 'Go there, I know not where; b
```



## Backward Search

Let $P[0..m)$ be a pattern and let $[b..e)$ be the suffix array range corresponding to suffixes that begin with $P$, i.e., $SA[b..e)$ contains the starting positions of $P$ in the text $T$. Earlier we noted that $[b..e)$ can be found by *binary search* on the suffix array.

Backward search is a different technique for finding this range. It is based on the observation that $[b..e)$ is also the range of rows in the matrix $\mathcal{M}$ beginning with $P$.

Let $[b_i, e_i)$ be the range for the pattern suffix $P_i$. The backward search will first compute $[b_{m-1}, e_{m-1})$, then $[b_{m-2}, e_{m-2})$, etc. until it obtains $[b_0, e_0) = [b, e)$. Hence the name backward search.



Backward search uses the following data structures:

- An array $C[0..\sigma)$, where $C[c] = \big|\{i \in [0..n] \mid L[i] < c\}\big|$. In other words, $C[c]$ is the number of occurrences of symbols that are smaller than $c$.
- The function $\mathit{rank}_L : \Sigma \times [0..n+1] \to [0..n]$:

  $$\mathit{rank}_L(c, j) = \big|\{i \mid i < j \text{ and } L[i] = c\}\big| \ .$$

  In other words, $\mathit{rank}_L(c, j)$ is the number of occurrences of $c$ in $L$ before position $i$.

Given $b_{i+1}$, we can now compute $b_i$ as follows. Computing $e_i$ from $e_{i+1}$ is similar.

- $C[P[i]]$ is the number of rows beginning with a symbol smaller than $P[i]$. Thus $b \geq C[P[i]]$.
- $\mathit{rank}_L(P[i], b_{i+1})$ is the number of rows that are lexicographically smaller than $P_{i+1}$ and contain $P[i]$ at the last column. Rotating these rows one step to the right, we obtain the rotations of $T$ that begin with $P[i]$ and are lexicographically smaller than $P_i$.
- Thus $b_i = C[P[i]] + \mathit{rank}_L(P[i], b_{i+1})$.

**Algorithm 4.24:** Backward Search
Input: array $C$, function *rank*$_L$, pattern $P$
Output: suffix array range $[b..e)$ containg starting positions of $P$

(1) $b \leftarrow 0$; $e \leftarrow n + 1$
(2) *for* $i \leftarrow m - 1$ *downto* 0 *do*
(3) $\quad c \leftarrow P[i]$
(4) $\quad b \leftarrow C[c] +$ *rank*$_L(c, b)$
(5) $\quad e \leftarrow C[c] +$ *rank*$_L(c, e)$
(6) *return* $[b..e)$

- The array $C$ requires an integer alphabet that is not too large.

- The trivial implementation of the function *rank*$_L$ as an array requires $\Theta(\sigma n)$ space, which is often too much. There are much more space efficient (but slower) implementations. There are even implementations with a size that is close to the size of the *compressed text*.