Re-Pair

Re-Pair is a greedy grammar compression algorithm that operates as follows:

1. Find the pair of symbols XY that is the most frequent in the text T. If no pair occurs twice in T, stop.
2. Create a new non-terminal Z and add the rule $Z \to XY$ to the grammar. Replace all occurrences of XY in T by Z. Go to step 1.

The whole process can be performed in *linear time* using suitable data structures. The details are omitted, but the key observation is that, if n_{XY} is the number of occurrences of the most frequent pair XY in a given step, then the replacement reduces the size of the grammar by $n_{XY} - 2$. Thus we can spend $\mathcal{O}(n_{XY})$ time to perform the step.

Here is a simple encoding of the final result:

- The number r of rules and the length z of the compressed text in γ code.
- The right-hand sides of rules using $\lceil \log(\sigma + i - 1) \rceil$-bit fixed length codes to encode the ith rule.
- The compressed text using $\lceil \log(\sigma + r) \rceil$-bit fixed length codes.

Better compression can be achieved with a more sophisticated encoding.

Example 2.13: $T =$ singing␣do␣wah␣diddy␣diddy␣dum␣diddy␣do.

rule added	text after replacement
$A \to$ ␣d	singing$A$o␣wah$A$iddy$A$iddy$A$um$A$iddy$A$o
$B \to$ dd	singing$A$o␣wah$A$iByAiByAum$A$iByAo
$C \to A$i	singing$A$o␣wahCByCByAumCByAo
$D \to B$y	singing$A$o␣wah$CDCDA$umCDAo
$E \to CD$	singing$A$o␣wahEEAumEAo
$F \to$ in	sFgFgAo␣wahEEAumEAo
$G \to A$o	sFgFgG␣wahEEAumEG
$H \to F$g	sHHG␣wahEEAumEG

A common feature of most dictionary compression algorithms is asymmetry of compression and decompression:

- The compressor needs to do a lot of work in choosing the phrases or rules.
- The decompressor only needs to replace each phrase.

Thus the decompressor is often simple and fast.

- LZ77-type methods are particularly simple and fast as they have no dictionary other than the already decoded part of the text.
- LZ78-type and grammar-based methods need some extra effort in constructing and accessing the dictionary.

The best possible compression may require a complex model and arithmetic coding to encode the phrase, which can make the decoder dramatically slower. Thus many implementations are optimized more for speed than maximum compression.

Burrows–Wheeler Transform

The Burrows–Wheeler transform (BWT) is a transformation of the text that makes it easier to compress. It can be defined as follows:

- Let $T[0..n)$ be a text. For any $i \in [0..n)$, $T[i..n)T[0..i)$ is a *rotation* of T. Let $\mathcal{M}$ be the $n \times n$ matrix, where the rows are all the rotations of T in lexicographical order.
- All columns of $\mathcal{M}$ are *permutations* of T. In particular:
  - The first column F contains the text symbols in order.
  - The last column L is the BWT of T.

Example 2.14: The BWT of $T =$ banana$ is $L =$ annb$aa.

F						L
$	b	a	n	a	n	a
a	$	b	a	n	a	n
a	n	a	$	b	a	n
a	n	a	n	a	$	b
b	a	n	a	n	a	$
n	a	$	b	a	n	a
n	a	n	a	$	b	a

Surprisingly, BWT is *invertible*, i.e., T can be reconstructed from the BWT L. We will later see how.

The compressibility of the BWT is based on sorting by context:

- Consider a symbol s that is followed by a string w in the text. (The text is considered to be cyclic.) The string w is called a *right context* of s. In the matrix $\mathcal{M}$, there is a row beginning with w and ending with s.
- Because the rows are sorted, all symbols with the right context w appear consecutively in the BWT. This part of the BWT sharing the context w is called a *w-context block* and is denoted by L_w.
- Context blocks are often highly compressible as they consist of symbols occurring in the same context.

Example 2.15: In Example 2.14, $L_{\mathtt{n}} = \mathtt{aa}$.

Here we have right contexts while earlier with higher order compression we considered left contexts. This makes no essential difference. Furthermore, the text is often reversed before computing the BWT, which turns left contexts into right contexts and vice versa.

Example 2.16: The context block $L_{\mathtt{ht}}$ for a reversed English text, which corresponds to left context `th` in unreversed text.

```
oreeereoeeieeeeaooeeeeeeaereeeeeeeeeeeeereeeeeeeeeeaaeeaeeeeeeeee
eaeeeeeeeeaeieeeeeeeeereeeeeeeeeeeeeeeeeeeeeeeeeaeeieeeeeeeaaieee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeaeieeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeaee
eeeeeeeeeeeeeeeeeereeeeeeeeeeieaeeeeieeeeaeeeeeeeeeeieeeeeeeee
eeeieeeeeeeeioaaeeaoereeeeeeeeeeeaaeaaeeeeieeeeeeeieeeeeeeeeaeeee
eeaeeeeeeereeeaeeeeeieeeeeeeeeiieee. e  eeeeiiiiii e          ,
i   o       oo e  eiiiiee,er  ,   ,      ,  . iii
```

Some of those symbols in (unreversed) 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
```


The context blocks are closely related to empirical entropies.

Theorem 2.17: For any text T and any k,

$$\sum_{w \in \Sigma^k} |L_w| H_0(L_w) = |T| H_k(T).$$

Proof. Let us first note that the equation for the kth order entropy proven in Exercise 3.4 is symmetric. Thus it does not matter whether we talk about the text or its reversal, or about the left or the right context.

Recall that n_w is the number of occurrences of w in the text. Then $n_w = |L_w|$ and

$$H_0(L_w) = -\sum_{s \in \sigma} \frac{n_{sw}}{n_w} \log \frac{n_{sw}}{n_w} = -\sum_{s \in \sigma} \frac{n_{ws}}{n_w} \log \frac{n_{ws}}{n_w}.$$

The strings L_w, $w \in \Sigma^k$, are distinct, i.e, they form a partitioning of the BWT into σ^k blocks (some of which may be empty). Furthermore,

$$\sum_{w \in \Sigma^k} |L_w| H_0(L_w) = -\sum_{w \in \Sigma^k} n_w \sum_{s \in \sigma} \frac{n_{ws}}{n_w} \log \frac{n_{ws}}{n_w} = n H_k(T).$$

$\square$

According to the theorem, zeroth order compression of the context blocks achieves kth order compression of the text. This is known as *compression boosting*.

As we noted earlier, using a single value of k everywhere is not optimal in general. There exists a linear time algorithm for finding a sequence of variable length contexts $w_1, w_2, \ldots, w_h$ such that $L_{w_1} L_{w_2} \ldots L_{w_h} = L$ and the total compressed size is minimized. This is called *optimal compression boosting*.

For the best compression, we may need to take multiple context lengths into account. With BWT this is fairly easy using adaptive methods:

- For most symbols s in the BWT, the nearest preceding symbols share a long context with s, symbols that are a little further away share a short context with s, and symbols far away share no context at all.
- Thus adaptive methods that forget, i.e., give higher weight to the nearest preceding symbols are often good compressors of the BWT.
- Such methods can be *context oblivious*: They achieve good compression for all contexts without any knowledge about context blocks or context lengths.

Run-length encoding

An extreme form of forgetting is run-length encoding (RLE). RLE encodes a run s^k of k consecutive occurrences of a symbol s by the pair $\langle s, k\rangle$. Nothing beyond the run has an effect on the encoding.

Example 2.18: The run-length encoding of $L_{\mathtt{ht}}$ from Example 2.16 begins: $\langle \mathtt{o}, 1\rangle$ $\langle \mathtt{r}, 1\rangle$ $\langle \mathtt{e}, 3\rangle$ $\langle \mathtt{r}, 1\rangle$ $\langle \mathtt{e}, 1\rangle$ $\langle \mathtt{o}, 1\rangle$ $\langle \mathtt{e}, 2\rangle$ $\langle \mathtt{i}, 1\rangle$ $\langle \mathtt{e}, 4\rangle$ $\langle \mathtt{a}, 1\rangle$ $\langle \mathtt{o}, 2\rangle$ $\langle \mathtt{e}, 5\rangle$.

RLE can be wastful when there are many runs of length one. A simple optimization is to encode the (remaining) run-length only after two same symbols in a row. This could be called lazy RLE.

Example 2.19: The lazy RLE of $L_{\mathtt{ht}}$ from Example 2.16 begins: `oree1reoee0iee2aoo0ee3`.

RLE alone does not compress the BWT very well in general but can be useful when combined with other methods.

Move-to-front

Move-to-front (MTF) encoding works like this:

- Maintain a list of all symbols of the alphabet. A symbol is encoded by its position on the list. Note that the size of the alphabet stays the same.
- When a symbol occurs, it is moved to the front of the list. Frequent symbols tend to stay near the front of the list and are therefore encoded with small values.

After an MTF encoding of the BWT, the smallest values tend to be the most frequent ones throughout the sequence. Thus a single global model can achieve a good compression. This is what makes MTF encoding context oblivious.

Example 2.20: The MTF encoding of $L =$ `annb$aa` is 0202330.

list	next symbol	code
abn$	a	0
abn$	n	2
nab$	n	0
nab$	b	2
bna$	$	3
$bna	a	3
a$bn	a	0

There are also other techniques that transform the BWT into a sequence of numbers.

Whatever the final sequence is — the plain BWT, an RLE encoded BWT, an MTF encoded BWT, or something else — it needs to be compressed using an entropy encoder to achieve the best compression. However, the model needed is much simpler than what would be needed for direct encoding of the text. For example:

- The plain BWT can be compressed well using a single zeroth order adaptive model. For a similar compression rate, the text needs to be encoded using a complex higher order model.
- The MTF encoded BWT can be compressed well using a single zeroth order semiadaptive model. An equivalent direct text compression would need a higher order semiadaptive model.

Computing and inverting BWT

Let us assume that the last symbol of the text $T[n-1] =$ $ does not appear anywhere else in the text and is smaller than any other symbol. This simplifies the algorithms.

To compute the BWT, we need to sort the rotations. With the extra symbol at the end, sorting rotations is equivalent to sorting suffixes. The sorted array of all suffixes is called the *suffix array* (SA).

F						L
$	b	a	n	a	n	a
a	$	b	a	n	a	n
a	n	a	$	b	a	n
a	n	a	n	a	$	b
b	a	n	a	n	a	$
n	a	$	b	a	n	a
n	a	n	a	$	b	a

SA							
6	$						
5	a	$					
3	a	n	a	$			
1	a	n	a	n	a	$	
0	b	a	n	a	n	a	$
4	n	a	$				
2	n	a	n	a	$		

There are linear time algorithms for suffix sorting. The best ones are complicated but fairly fast in practice. We will not described them here, but they are covered on the course *String Processing Algorithms*.

We will take a closer look at inverting the BWT, i.e., recovering the text T given its BWT.

Let $\mathcal{M}'$ be the matrix obtained by rotating $\mathcal{M}$ one step to the right.

Example 2.21:

$$\begin{array}{ccccccc|c|} \multicolumn{7}{c}{\mathcal{M}} \\ \cline{7-7} \$ & b & a & n & a & n & a \\ a & \$ & b & a & n & a & n \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ b & a & n & a & n & a & \$ \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \\ \cline{7-7} \end{array} \xrightarrow{\text{rotate}} \begin{array}{|c|cccccc} \multicolumn{7}{c}{\mathcal{M}'} \\ \hline a & \$ & b & a & n & a & n \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \\ b & a & n & a & n & a & \$ \\ \$ & b & a & n & a & n & a \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ \hline \end{array}$$

- The rows of $\mathcal{M}'$ are the rotations of T in a different order.
- In $\mathcal{M}'$ without the first column, the rows are sorted lexicographically. If we sort the rows of $\mathcal{M}'$ *stably* by the first column, we obtain $\mathcal{M}$.

This cycle $\mathcal{M} \xrightarrow{\text{rotate}} \mathcal{M}' \xrightarrow{\text{sort}} \mathcal{M}$ is the key to inverse BWT.

- In the cycle, each column moves one step to the right and is permuted. The permutation is fully determined by the last column of $\mathcal{M}$, i.e., the BWT.
- By repeating the cycle, we can reconstruct $\mathcal{M}$ from the BWT.
- To reconstruct T, we do not need to compute the whole matrix just one row.

Example 2.22:

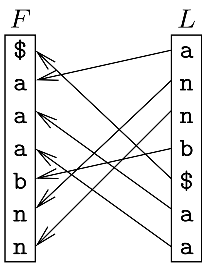
```
------a          a------          $-----a          a$-----          $b----a
------n          n------          a-----n          na-----          a$----n
------n  rotate  n------   sort   a-----n  rotate  na-----   sort   an----n
------b   -->    b------   -->    a-----b   -->    ba-----   -->    an----b
------$          $------          b-----$          $b-----          ba----$
------a          a------          n-----a          an-----          na----a
------a          a------          n-----a          an-----          na----a

          $ba---a            $ban--a            $bana-a            $banana
          a$b---n            a$ba--n            a$ban-n            a$banan
rotate    ana---n  rotate    ana$--n  rotate    ana$b-n  rotate    ana$ban
& sort    ana---b  & sort    anan--b  & sort    anana-b  & sort    anana$b
 -->      ban---$   -->      bana--$   -->      banan-$   -->      banana$
          na$---a            na$b--a            na$ba-a            na$bana
          nan---a            nana--a            nana$-a            nana$ba
```


The permutation that transforms $\mathcal{M}'$ into $\mathcal{M}$ is called the *LF-mapping*.

- LF-mapping is the permutation that stably sorts the BWT L, i.e., $F[LF[i]] = L[i]$. Thus it is easy to compute from L.
- Given the LF-mapping, we can easily follow a row through the permutations.

Example 2.23:



Here is the algorithm.

Algorithm 2.24: Inverse BWT
Input: BWT $L[0..n]$
Output: text $T[0..n]$
Compute LF-mapping:
  (1) *for* $i \leftarrow 0$ *to* n *do* $R[i] = (L[i], i)$
  (2) sort R (stably by first element)
  (3) *for* $i \leftarrow 0$ *to* n *do*
  (4) $\quad (\cdot, j) \leftarrow R[i];\ LF[j] \leftarrow i$
Reconstruct text:
  (5) $j \leftarrow$ position of \$ in L
  (6) *for* $i \leftarrow n$ *downto* 0 *do*
  (7) $\quad T[i] \leftarrow L[j]$
  (8) $\quad j \leftarrow LF[j]$
  (9) *return* T

The time complexity is linear if we sort R using counting sort.