Consider the following text: *The man was very ol_.* Only a limited number of characters in the alphabet can be sensibly placed in the blank. This implies that the number of bits needed to encode the phrase is smaller than if each character was encoded independently. A compression scheme cannot take advantage of English language semantics, but it turns out that context-based information can improve the compression preformance. Below we describe four *context-based* algorithms.

# 1 Run-length encoding

Represent a string by replacing each subsequence of consecutive identical characters with $(char, length)$. The string 11112222333111 would have representation $(1,4)(2,4)(3,3)(1,3)$. Then compress each $(char, length)$ as a unit using, say, Huffman coding. Clearly, this technique works best when the characters repeat often. One such situation is in fax transmission, which contains alternating long sequences of 1's and 0's. The distribution of codewords is taken over many documents to compute the optimal Huffman code.

# 2 Move-to-front encoding

This is a technique that is ideal for sequences with the property that the occurrence of a character indicates it is more likely to occur immediately afterwards. We convert a sequence of characters to a list of numbers as follows: We maintain a list of characters and represent characters by their position in the list. On encoding a character, it is moved to the front of the list. Thus smaller numbers are more likely to occur than larger numbers.

Consider the sequence *abaaccd* as an example. The initial working list of characters has characters in some order, say $(a, b, c, d)$. The first $a$ would be represented by a 1. The next character, $b$, is the second symbol in $(a, b, c, d)$ so we represent it by a 2 and move the second symbol in the list to the front. The list is now $(b, a, c, d)$. The next character $a$ is represented by a 2, the next $a$ by a 1, $c$ by 3, the next $c$ by 1 and $d$ by 4.

Run-length and move-to-front encoding are not used alone very much but are useful as subprocedures in other techniques.

# 3 Prediction by partial matching (PPM)

Main idea: generate a conditional probability for the current character based on the previous characters. To show by example, consider *chillie_*. The underscore represents the current character, which we want to guess. In a known text database, the character sequence *illie* does not appear, *llie* appears once followed by a $s$, and *lie* appears 201 times, 19 of these followed by a $r$. The below diagram illustrates this example.

$$\begin{tikzcd}
llie_\ (1) \arrow[r, "\text{escape symbol}"] \arrow[d] & lie_\ (201) \arrow[r, "\text{escape symbol}"] \arrow[d] & \cdots \\
s:1 & r:19 \\
 & \cdots
\end{tikzcd}$$

The strings *illie* and *llie* are *contexts*, or substrings to be matched in the text. Assume that the algorithm starts with contexts of length 4. It can select the match with $s$ and stop. Or, it can output an *escape character* and consider the next smaller context (here, of length 3) and so on.

Variants of PPM differ in how they assign probabilities to matching symbols or an escape symbol. Let $c_i$ represent the frequency of character $i$ that follows the current context in the text database, and $r$ the number of distinct symbols seen in the context.

| $Variant$ | $Pr(escape)$ | $Pr(c_i)$ |
|---|---|---|
| PPMA | $\frac{1}{\sum c_i+1}$ | $\frac{c_i}{\sum c_i+1}$ |
| PPMC | $\frac{r}{\sum c_i+r}$ | $\frac{c_i}{\sum c_i+r}$ |
| PPMD | $\frac{r}{2\sum c_i}$ | $\frac{2c_i-1}{2\sum c_i}$ |

PPM* is like PPMD, but uses arbitrarily large contexts, compared to those of a fixed maximum length for the other versions.

# 4 Burrows-Wheeler

Burrows-Wheeler is a relatively new algorithm implemented as `bzip`. We show the encoding portion by example on the string *mississippi*. First, associate each character in the string with its context. Below, the first box shows each character with its context. Second, sort each context in right-to-left fashion. The second box shows the rows in sorted order.

| | |
|---|---|
| `ississippi m` | `sissippimi s` |
| `ssissippim i` | `ississippi m` |
| `sissippimi s` | `sippimissi s` |
| `issippimis s` | `pimississi p` |
| `ssippimiss i` | `ssissippim i` |
| `sippimissi s` | `imississip p` |
| `ippimissis s` | `mississipp i` |
| `ppimississ i` | `issippimis s` |
| `pimississi p` | `ippimissis s` |
| `imississip p` | `ssippimiss i` |
| `mississipp i` | `ppimississ i` |

The last column is the output, and the most significant character in each context is the rightmost one. Call the output column $A$ and the most-significant column $B$. Essentially, what Burrows-Wheeler does is to convert the initial sequence into one that is good for move-to-front encoding, which is then applied.

**Claim 4.1** *The original sequence can be reconstructed from the first item and columns $A$ and $B$.*

The key is that characters of the same value appear in the same order in the output column and the most-significant column. In this example, the first character is $m$, which is the first $m$ in column $A$. To find the second character, look up the first $m$ in column $B$. The column $A$ value for that row is the first $i$. Now look up the first $i$ in column $B$, etc.

*Running time.* For a string of length $n$, use $n$ buckets (assuming that the size of the character set is about $n$). Then each iteration requires $n$ time. There are $n$ iterations, so $O(n^2)$ worst case. Doubling in each iteration, i.e. sorting context by length $i$ suffixes, allows only $\log n$ iterations, or $O(n \log n)$ time.