10.3 Multi-character encodings

So far, we have studied single-character encodings, which consist of an assignment of code-words to characters. However, in real life it makes sense to encode longer strings with one code-word if there are some strings that repeat frequently. For example, in English text we will frequently have th, an, or longer substrings such as the, then and and. Likewise, in HTML we will frequently have <a href, "<img src", "<br>", and in C++ we will frequently have for, while, etc. It makes sense to assign a single code-word to such strings.

This raises, however, two problems:

• How many code-words should we use? The more code-words we use the longer the individual code-words must be. But on the other hand, more code-words mean that longer strings can be encoded.

• How should we determine which strings should obtain a code-word? In particular, this is highly dependent on the language of the encoded text. Ideally, the encoding scheme would determine this by itself, without user help.

We will see two examples multiple-character encodings here. The first one (run-length-encoding) is very limited, and really only suitable for special situations. On the other hand, the second one (Lempel-Ziv-Welch encoding) is very versatile and performs very well on English text.

10.3.1 Run-Length Encoding

We assume throughout this section that both the source text and the compressed texts are bit-strings, i.e., $\Sigma_S = \Sigma_C = \{0, 1\}$. The main idea of run-length encoding is that we can describe any bit-string by listing its first character, and listing the lengths of its runs (where a run is a maximal set of consecutive characters that are the same). Consider for example the source text

$S = \underline{00000} \underline{111} \underline{0000}.$

We can describe $S$ as “0,5,3,4”, which means the following:

• The first bit of $S$ is 0.

• The first run of $S$ has length 5, i.e., $S$ begins with 00000.

• The next run of $S$ has length 3. Since runs were defined to be maximal subsequences of identical characters, and since the alphabet is $\{0, 1\}$, we know that this run necessarily consists of 1s. Therefore, the next 3 characters of $S$ are 111.

• The next run of $S$ has length 4. By a symmetric argument therefore the next 4 characters of $S$ are 0000.

Reversing these steps, one can easily deduce from encoded sequence 0,5,3,4 that the source text was 000001110000, so we can recover the source text from the sequence of integers.
Encoding sequences of integers

There is only one hitch: Currently, we describe the output as a list of integers, but our goal was to have a bit-string as compressed string. This therefore raises the question: how to encode an integer as a bit-string? Any one integer is naturally encoded as bit-string by using its base-2 representation, but if we want a sequence of integers, then we either must allow a third character (a comma or some similar separator), or find a way to express a comma through a special bit-sequence.

We describe here the Elias-gamma-code that can be used for a lossless encoding of a sequence of positive integers. First, encode one integer \( k > 0 \) as follows:

- Write \( \lfloor \log k \rfloor \) copies of 0.
- Then write the binary representation of \( k \) (using the minimal number of bits, which in particular implies that it always starts with 1).

See Table 10.1 for the first few encodings. If we use \( E(k) \) for the encoding of integer \( k \), then we can encode a sequence of positive integers by concatenating their encodings, i.e., \( k_1, k_2, k_3, k_4 \) becomes \( E(k_1)E(k_2)E(k_3)E(k_4) \). We will argue below (as part of run-length encoding) that this is uniquely decodable, even though the resulting bit-string contains no comma or other indicator to say where one encoding ends and the next one begins.

Run-length encoding details

The run-length encoding of a bit-string \( S \) is now obtained as follows:

- Initialize \( C \) with the first bit of \( S \).
- Determine the run-lengths \( k_1, k_2, \ldots, k_d \) of \( S \).
- Append \( E(k_1), \ldots, E(k_d) \) to \( C \), in order.

<table>
<thead>
<tr>
<th>( k )</th>
<th>( \lfloor \log k \rfloor )</th>
<th>( k ) in binary</th>
<th>encoding ( E(k) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>10</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>11</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>100</td>
<td>00100</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>101</td>
<td>00101</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>110</td>
<td>00110</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
</tbody>
</table>

Table 10.1: Elias-gamma-codes for positive integers.
Algorithm 1: Computing the run-length encoding.

Input: Source text \( S[0..n-1] \) (a bit-string)
Initialize output stream \( C \); \( C.append(S[0]) \)
\( i \leftarrow 0 \) // index of current char of \( S \)

while \( i < n \) do

\( k \leftarrow 1 \) // length of run

while \( i + k < n \) and \( S[i + k] = S[i] \) do
\( i \leftarrow i + k \)

\( K \leftarrow \) empty string // binary encoding of \( k \)

while \( k > 1 \) do

\( C.append(0) \)

\( K.prepend(k \text{ mod } 2) \)

\( k \leftarrow \lfloor k/2 \rfloor \)

\( K.prepend(1) \)

\( C.append(K) \)

return \( C \)

Algorithm 1 gives the pseudo-code for run-length encoding. The run-time of this algorithm is \( O(|S| + |C|) \), because with every step we either advance one bit in \( S \), or we append one or more bits to \( C \). Let’s consider a few examples:

- \( 00000 \underbrace{111 \ldots 000}_{5 \text{ ones}} \) becomes \( 0 \underbrace{001 \ldots 0}_{3 \text{ ones}} 111 \underbrace{000}_{4 \text{ ones}} \)

(\( E(5) \underbrace{E(3) E(5)}_{5 \text{ binary bits}} \))

(the “not-under-braced” 0 is the leading bit—don’t forget that!)

- \( 111111110100000000000000000000000001111111111 \) can be broken into runs as follows:

\( 1 \underbrace{1111111}_{7 \text{ ones}} 00 \underbrace{100000000000000000000000000000000001111111}_{20 \text{ ones}} \)

and therefore becomes encoded as

\( 1 \underbrace{001111010}_{7 \text{ ones}} 1 \underbrace{000010100}_{20 \text{ ones}} 0001011 \underbrace{E(11)}_{E(11) \text{ bits}} \)

- \( 111111111111 \ldots 1 \) becomes \( 1 \underbrace{00000 \ldots 0}_{\log n \text{ ones}} \underbrace{xxxx \ldots xxxx}_{(\log n)-1 \text{ ones}} \)

In particular, we use \( O(\log n) \) bits to encode \( n \) characters in this special case. This is (by the information-theoretic lower bound) asymptotically as small as possible.
Run-length decoding

Recall that (for this section) the binary encoding of a positive integer \( k \) always starts with a 1; we omit all leading 0s. This is exactly what makes decoding feasible: for as long as we see 0s, we know that this is the “counting” part of the Elias-gamma-code that tells us how long the binary string is going to be. So to decode a run-length encoding, we proceed as follows:

- Read the initial bit.
- For as long as there are 0s, keep increasing a counter \( \ell \).
- Get the next \( \ell + 1 \) bits to get the binary encoding of the run-length.
- Create the appropriate run (we know whether it consists of 0s or 1s from the initial bit, which we flip with every round).
- Append the run to the output and repeat.

Algorithm 2 gives the pseudocode for this procedure. The run-time of this algorithm is \( O(|S| + |C|) \), because with every step we either advance one bit in \( C \), or we append one more bits to \( S \). Note that this algorithm can fail if the input was not appropriate, i.e., not a run-length encoding of a string: we want to get \( \ell + 1 \) bits from the input, and if there are not sufficiently many bits left in the input then this was not valid.\(^1\)

For example, when given the bit-string \( C = 00001101001001010 \), we

- first extract the initial bit (‘0’),
- now read three more 0s, which tells us that the binary encoding of the next run has 4 bits,
- now we extract the next 4 bits so that we get the first Elias-gamma-code:

\[
C = 00001101001001010 \quad \underbrace{E(k)}_{13}
\]

This decodes to 13, so we know that the output \( S \) starts with 13 0s.

- Repeating this, we get three more encodings of runs:

\[
C = 00001101001001010 \quad \underbrace{1010}_{E(1) E(2)}
\]

\[^1\text{If this happens to you during the exam, then most likely you have miscounted somewhere, or forgotten to remove the leading bit first, or forgotten that the length of the 0-run is one less than the length of the binary encoding. Re-start!}\]
Algorithm 2: Decoding a run-length encoding.

Input: Compressed text \( C \) as a stream of bits

Initialize output stream \( S \)

\( b \leftarrow C.pop() \)  // bit-value for the current run

\( \textbf{while } C \text{ has bits left do} \)

\( \ell \leftarrow 0 \)  // length of binary encoding of run, minus 1

\( \textbf{while } C.pop() = 0 \text{ do} \)

\( \ell++ \)

\( k \leftarrow 1 \)  // run-length

\( \textbf{for } j \leftarrow 1 \text{ to } \ell \text{ do} \)

\( \text{if } C \text{ has no bits left then} \)

\( \text{return } \text{“invalid encoding”} \)

\( \text{else} \)

\( k \leftarrow k \times 2 + C.pop() \)

\( \textbf{for } j \leftarrow 1 \text{ to } k \text{ do} \)

\( S.append(b) \)

\( b \leftarrow 1 - b \)  // flip bit for next run

So the run-lengths are 13,4,1,2, and since we start with 0 and alternate the bits, we have

\[ S = \underbrace{0000000000000}_{13} \underbrace{1111}_{4} \underbrace{0}_{1} \underbrace{11}_{2} \]

Run-length encoding summary

Run-length encoding is a very simple and very fast method of converting a bit-string into a bit-string. The problem with it is that its compression ratio depends very much on the input and the run-lengths that it has. A run of \( k \) bits is compressed to \( 2 \log k + 1 \in o(k) \) bits. This is great if \( k \) is big, but if \( k \) is small then this is not good. Indeed, we have no compression unless \( k \geq 7 \), and for \( k = 2 \) or 4 the Elias-gamma-code uses more than \( k \) bits. In consequence, if there are lots of runs of length up to 6 and few longer runs, then the run-length-encoding is longer than the original!

The one place where RLE has proven useful is in transmitting pictures, especially black-and-white pictures. Imagine a picture represented by using 0 for white and 1 for black. Most pictures have large patches of all-white or all-black, which corresponds to long runs, so for encoding pictures the compression ratio achieved with RLE should be good. This is especially true if the picture is of a piece of paper with a few words on them, where nearly everything is white. In particular, RLE was very popular with fax-machines (back when we still used such things). These days it is used in some image formats (TIFF), and as part of
One drawback of RLE is that it crucially requires the input to be a bit-string. One can of course convert any input into a bit-string, for example by converting every ASCII-character into its 7-bit representation. However, this would destroy all the runs. Therefore, to use RLE for arbitrary alphabets, we must send not only the length of the run but also which character it encodes. For example, the word $BBBAACC$ could be described by $B, 3, A, 2, C, 2$ (which then in turn we must encode somehow, e.g. using Elias-gamma-codes).

### 10.3.2 Lempel-Ziv-Welch

We now turn to the Lempel-Ziv-Welch (or LZW) encoding, which also represents longer strings of characters with one code-word, but differs from RLE in multiple ways:

- It does not need to know what substring should get encoded by one code-word. (This is in contrast to RLE, where we fixed that only long runs of repeating characters get encoded by one code-word.)

- It uses an adaptive dictionary, which means that the dictionary changes during the encoding. (This is in contrast to all previous encoding schemes, where the dictionary was static: The dictionary was fully determined before encoding starts and never changed afterwards.)

Allowing an adaptive dictionary may sound a little scary—don’t we have to send the dictionary to the decoder then so that he/she knows how to decode? The trick for using adaptive dictionaries is to create a rigid set of rules of how the dictionary is changed. Furthermore, the rules must be chosen in such a way that the decoder can deduce how the dictionary is changed. Put differently, we do not send the dictionary along, but there is a fixed way of how it was created, and the decoder can re-create the exact same dictionary during the decoding process. Needless to say, this makes decoding less obvious than it has been in the past!

### LZW encoding

For now, we will explain LZW as mapping a string of ASCII characters into a list of non-negative integers. There are numerous ways of mapping that list into a bit-string, we will discuss this further later.

So assume we have a string $S$ of ASCII characters. Since we have an adaptive encoding scheme, we also maintain a dictionary $D$ that maps strings to their code-words (which are non-negative integers). Initially, $D$ is simply ASCII itself (recall that each ASCII-character naturally corresponds to a non-negative integer). The LZW-encoding now consists of only two steps, repeated over and over until the entire string $S$ is processed:
Algorithm 3: Lempel-Ziv-Welch encoding

Input: Text $S$ as a stream of ASCII-characters
Initialize dictionary $D$ as a trie that maps ASCII to $\{0, \ldots, 127\}$

\[ idx \leftarrow 128 \] // global counter for first free code-number

while $S$ has characters left do
  \[ v \leftarrow \text{root of trie } D \]
  \[ K \leftarrow S.\text{peek}() \]
  \[ v \leftarrow c.S.\text{pop}() \]
  \[ \text{if there is no more input in } S \text{ then} \]
  \[ \text{break} \]
  \[ K \leftarrow S.\text{peek}() \]
  output code-number stored at $v$
  if $S$ has characters left then
    create child of $v$ labelled $K$ with code-number $idx$
    $idx++$

Based on what we need to do with $D$, we see that the best way to store it is in a trie. Every node of the trie (except the root) will store a code-number, and the code-number at node $v$ corresponds to the string of characters on the edges to $v$. Finding the string $w$ is then very easy: we simply parse the characters of $S$, starting from the root, until we reach “no such child” at some node $v$ for some character $K$; the code-number to use is then simply the one stored at $v$. Adding the new entry in $D$ is also very easy—simply add the child at $v$ that we would have liked to have (i.e., with character $K$) and give it the next available code-number, which we maintain with a global counter. Algorithm 3 gives the details. Note that we have $O(1)$ run-time per character that was removed from $S$, so the run-time of this algorithm is $O(|S|)$.

Figure 10.1 shows an example of how the encoding works. This can be read as follows: The vertical line segments denote the end of the successfully read string $w$. The larger number in the interval between two such segments is the code-number that corre-
sponds to \( w \). The smaller number above the segment is the code-number that has been assigned in this round. The string that is assigned to it consists of the entire string on the left, and the first character of the string on the right. (Sometimes we will indicate this string with a dotted box.) The sequence of code-numbers that encodes this string hence is 65, 78, 128, 65, 83, 128, 129.

Let us do another example `barbarabarbarbaren`. In this example, we’ll omit the trie (it is useful for computers, but for humans on our small examples it is not really needed—we can scan the existing codes fast enough to see the longest string that fits). All we need to know is the ASCII numbers for the characters that are in our text. See Figure 10.2. Notice how quickly LZW “clued in” that the string `bar` is very important here and should have a code-number assigned to it.

```
ba | ar | rb | bar | ra | ab | 128 | 129 | 130 | 131 | 132 | 133 | 134 | 135 | 136 | 137
b  | a  | r  | b  | a  | r  | 98  | 97  | 114 | 128 | 114 | 97  | 131 | 134 | 129 | 101
```

Figure 10.2: Second example for LZW encoding.

To see how well LZW can compress strings in the best case, let us also consider the string \( a^n \), i.e., with \( n \) copies of the same character (in this case \( a \)). In the first round, we write code-number 97 for ‘a’ and assign 128 to ‘aa’. In the next round, we immediately use 128, and assign 129 to ‘aaa’. In the next round, we immediately use 129 and assign 130 to ‘aaaa’. Generally in the \( i \)th round (where initially \( i = 1 \)) we write the code-number for \( a^i \).

---

\(^2\)That’s part of a German tongue twister: rhabarbarbarabarbarbarenbartbarbierbier. Generally the texts that we use for compression get sillier and sillier, because they have to be chosen with lots of repetitions for the compression to do its magic.
This means that we encoded in total the string $a^I$ where $I = \sum_{j=1}^{i} j = \frac{i(i+1)}{2}$. This means that after roughly $\sqrt{2n}$ rounds, we have encoded the entire string. So in this case LZW uses $O(\sqrt{n})$ code-numbers. One can also argue that it cannot use fewer code-numbers, because the code-number used in round $i$ can encode at most $i$ characters. So the best case for LZW is worse than the best-case for RLE. However, in practice LZW far beats RLE since it detects suitable repeats, rather than relying on runs to exist in the text.

**LZW decoding**

Now let us turn to the problem of decoding a given sequence of integers that was obtained using LZW encoding. This is a lot harder than for previous encoding schemes, because LZW uses an adaptive dictionary and so we (the decoders) must also build this dictionary while decoding. Let us use as running example the following sequence of code-numbers:

$$C = 67, 65, 78, 32, 66, 129, 133, 83$$

A few of the code-numbers are ASCII (i.e., in the range 0-127), and for those we know what they stand for: the unknown source text $S$ begins with CAN BA. But to know what code-number 129 stands for, we really need to build the dictionary $D$. But we know how this was done: in each step, take the string that was encoded, and append the first character of the next string; the combination gets assigned to the next code-number. So we have

$$C \cdot A \rightarrow 128 \quad A \cdot N \rightarrow 129 \quad N \cdot \ U \rightarrow 130 \quad \ U \cdot B \rightarrow 131 \quad B \cdot A \rightarrow 132$$

which tells us that 129 decodes to AN. To be more systematic, let us put the decoding-steps into a table, listing in each row the current code-number, decoded string, and what was added to $D$. (The rightmost column will be explained in a little bit.)

<table>
<thead>
<tr>
<th>input</th>
<th>decodes to</th>
<th>Code #</th>
<th>String (human)</th>
<th>String (computer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>A</td>
<td>128</td>
<td>CA</td>
<td>67, A</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
<td>129</td>
<td>AN</td>
<td>65, N</td>
</tr>
<tr>
<td>32</td>
<td>(\ U )</td>
<td>130</td>
<td>(\ U )N</td>
<td>78, (\ U )</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
<td>131</td>
<td>(\ U )B</td>
<td>32, B</td>
</tr>
<tr>
<td>129</td>
<td>AN</td>
<td>132</td>
<td>BA</td>
<td>66, A</td>
</tr>
<tr>
<td>133</td>
<td>??</td>
<td>133</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the next step, however, we have a problem. We encounter code-number 133, but 133 is not yet in the dictionary! This happens because the decoder is “one step behind” in building the dictionary—it needs to know the first character of the next string to add to the dictionary, so can do this only once the next string is known, i.e., one step later.
One could think of some ways to fix this. For example, we could have imposed on the encoder not to use a code-number until at least one more step has gone by. But it turns out that this is not needed—by studying what happened during the encoding process, we can actually deduce what the string encoded by 133 is, even though it is not in the dictionary yet. Let us look at what happened in the encoding $S$ using $x_1, x_2, \ldots$ for characters of $S$ that we don’t know yet.

So we know that 133 encodes $ANx_1$; the only part we don’t know is what $x_1$ stands for. However, because we are in the special situation where 133 is used immediately after it has been added to the dictionary, we actually know what $x_1$ is. Look at the picture of encoding again. Because 133 was used in the next step, its corresponding string is $x_1x_2x_3$. But we also know that 133 encodes $ANx_1$. This means that $x_1$ must be $A$. More generally, whenever we encounter a code-number that is not yet in $D$ but about to be added, then it encodes the previous string, plus the first character of the previous string repeated. If we encounter a code-number that is even bigger, then the encoding was not valid. With this, we are ready for the pseudocode for LZW decoding, which is in Algorithm 4. Also, Figures 10.3 and 10.4 show the above example finished, and another example.

To analyze the run-time for Algorithm 4, observe that the while-loop executes $|C|$ times. Within one execution, we spend $O(|s|)$ time to append $s$ to $S$, making the total run-time $O(|C| + |S|) = O(|S|)$ when disregarding the time that it takes to look up code in dictionary.
Algorithm 4: Lempel-Ziv-Welch decoding

Input: Encoding $C$ as a stream of integers
Initialize $D$ as a dictionary that maps $\{0, \ldots, 127\}$ to ASCII
Initialize empty output stream $S$

$idx \leftarrow 128$  // global counter for first free code-number
$code \leftarrow C.pop()$  // first number creates no entry in $D$
$s \leftarrow LZW-dictionary-lookup(D, code)$
$S.append(s)$

while $C$ has codes left do
  $s_{prev} \leftarrow s$
  $code \leftarrow C.pop()$
  if $code < idx$ then
    $s \leftarrow LZW-dictionary-lookup(D, code)$
  else if $code = idx$ then  // special situation
    $s \leftarrow s_{prev} + s_{prev}[0]$
  else
    return "invalid encoding"
  $S.append(s)$
  insert $s_{prev} + s[0]$ into $D$ with code-word $idx$
  $idx++$

return $S$

The time to look up $code$ in $D$ depends on how $D$ is stored. The straightforward method of storing $D$ is to use an array that stores strings indexed by the code-number, as suggested in Figure 10.3. Then the lookup takes $O(1)$ time and returns a word. However, observe that the words stored in the dictionary might get quite long; even on our small example in Figure 10.4 we needed to store \texttt{barba}, a word of length 5. Moreover, it is unnecessary to store the entire word \texttt{barba}: We know that the word \texttt{barb} was also in $D$, because the only way to add a new word to $D$ is to take an existing word and append a character. So a much more space-efficient way to store a word in $D$ is to store where its prefix was, and what character $c$ was appended. Conveniently, we can refer to the prefix simply by its code-number, which allows us to look up its word recursively. See the rightmost column of Figure 10.3 for how dictionary $D$ is stored, and Algorithm 5 for how to look up a word in $D$.

This lookup takes more than constant time, but the time is proportional to $|s|$. Since we needed this time to append $s$ to the output anyway, this is only a constant overhead; the run-time for LZW decoding remains at $O(|S|)$. 
<table>
<thead>
<tr>
<th>input</th>
<th>decodes to</th>
<th>Code #</th>
<th>String (human)</th>
<th>String (computer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>b</td>
<td></td>
<td></td>
<td>98, a</td>
</tr>
<tr>
<td>97</td>
<td>a</td>
<td>128</td>
<td>ba</td>
<td>97, r</td>
</tr>
<tr>
<td>114</td>
<td>r</td>
<td>129</td>
<td>ar</td>
<td>114, b</td>
</tr>
<tr>
<td>128</td>
<td>ba</td>
<td>130</td>
<td>rb</td>
<td>128, r</td>
</tr>
<tr>
<td>97</td>
<td>a</td>
<td>132</td>
<td>ra</td>
<td>114, a</td>
</tr>
<tr>
<td>131</td>
<td>bar</td>
<td>133</td>
<td>ab</td>
<td>97, b</td>
</tr>
<tr>
<td>134</td>
<td>barb</td>
<td>134</td>
<td>barb</td>
<td>131, b</td>
</tr>
<tr>
<td>129</td>
<td>ar</td>
<td>135</td>
<td>barba</td>
<td>134, a</td>
</tr>
<tr>
<td>101</td>
<td>e</td>
<td>136</td>
<td>are</td>
<td>129, e</td>
</tr>
<tr>
<td>110</td>
<td>n</td>
<td>137</td>
<td>en</td>
<td>101, n</td>
</tr>
</tbody>
</table>

Figure 10.4: A second LZW decoding example. The special situation occurred when decoding 134.

Algorithm 5: LZW-dictionary-lookup

**Input:** Dictionary D as array of (int,char)-pairs, integer code

if code < 128 then // First 128 entries of D are ASCII-encoding
    Initialize output string s with D[code]
else // recursion
    (code<sub>prev</sub>, c) ← D[code]
    s ← LZW-dictionary-lookup(D, code<sub>prev</sub>)
    s.append(c)
return s

LZW discussion, history and variants

In our description of LZW thus far, we have used positive integers as encoding. This is problematic, partially because we wanted bit-strings for encodings, but more crucially because we are allowing an infinite alphabet \( \Sigma_C \) for the encoding (which is unfair since this has much more expressive power).

There are numerous ways of how to convert the LZW encoding into a bit-string. The simplest (and original) approach was to use a fixed-length encoding for the integers, for example encode every integer as a 12-bit bit-string. To give just one example, the output from Figure 10.1 would become

```
000001000001 000001001110 000010000000 000001000000 000001001001 000010000000 000010000000
```
The main disadvantage here is that we then have “only” 4096 code-numbers at our disposal (or actually only 3968 since the first 128 are reserved for ASCII). If we run out of code-numbers, then we must either stop adding to the dictionary, or re-assign some of the code-numbers. The assumption is that (at least for most languages) 4096 code-numbers are enough to capture most commonly used patterns, and adding more code-numbers (which comes at the price of adding more bits per code-number) is not worth it.

Of course there are other options. We just saw in RLE how to encode an arbitrary sequence of integers, using Elias-gamma-codes, into a bit-string that can be uniquely decoded. With this approach, we need not put a bound onto the code-numbers that we use. However, note that even the first code-number 128 (which has encoding $E(128) = 000000010000000$) uses 15 bits, and encoding the last code-number 4095 uses 23 bits. So the bit-strings of code-numbers are longer than 12 bits (except at the initial ASCII characters), and this is unlikely to achieve a better compression ratio.

There are lots of other variants of LZW. We currently start with dictionary $D$ initialized as ASCII, but one could instead allow a bigger dictionary, such as ISO-8859. We allowed LZW to immediately re-use a code-number that it has assigned; some other methods disallow this for ease of decoding. The most important part is that the encoder and decoder must agree on what exactly the rules are. The intent of LZW is that it does not send the dictionary along (in contrast to Huffman!), and instead it only sends the minimal information needed for the decoder to know what variant has been used to build $D$.

LZW encoding can be very bad in theory. If the input text has no repeated substrings, then an ASCII text (with 7 bits per character) gets effectively mapped to the same ASCII text (but now using 12 bits per character), so it may get longer! However, in practice there are lots of repeated substrings, and on English text a compression ratio (for the 12-bit-fixed-length encoding) of $\approx 50\%$ has been reported.

LZW encoding is also an interesting case study of the history of development of programs and the debate of patents for computer software. The original idea was invented by Lempel and Ziv in 1977 (with a major variation in 1978). In 1984, Welch published an improvement (the LZW encoding presented here). This was widely picked up by computer scientists and implemented in numerous standard compression algorithms, such as compress and GIF. Unbeknown to many of these users, multiple patents had been filed both on the Lempel-Ziv algorithms and on LZW. Major controversy erupted when Unisys (the holder of the patent on LZW) tried to enforce the patent in the 1990s and asked users to pay a license fee. Some users (such as Adobe, which used LZW for compression of postscript files) complied. But many others did not, and either used LZW-encoded files such as GIF illegally, or abandoned those file-formats and used others such as PNG (which has been developed precisely in response to the patent-issues). Nowadays, using GIF is no longer illegal (the patents expired in the early 2000s), but it has lost the “race” for predominance in picture-storing-techniques even though it was the better compression at the time.
### Transformation name

<table>
<thead>
<tr>
<th>Transformation name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Burrows-Wheeler transform</td>
<td>If $T_0$ has repeated longer substrings, then $T_1$ has long runs of characters.</td>
</tr>
<tr>
<td>Move-to-front transform</td>
<td>If $T_1$ has long runs of characters, then $T_2$ has long runs of zeros.</td>
</tr>
<tr>
<td>Modified RLE</td>
<td>If $T_2$ has long runs of zeroes, then $T_3$ has chars $A'$ and $B'$ very frequently</td>
</tr>
<tr>
<td>Huffman encoding</td>
<td>Compresses well since input-chars are unevenly distributed</td>
</tr>
</tbody>
</table>

Figure 10.5: The main steps of bzip2.

### 10.4 bzip2

Huffman encoding is very old (from the 1950s). Lempel-Ziv-Welch is newer, but also not new (from the 1970s and 1980s). Researchers are continually working to try to improve compression ratios even more on real-life texts. A real breakthrough came with the development of bzip2, which happened in the 1990s and 2000s. The crucial insight was the idea of a text transform: Before compressing the text, modify the text first (without changing its length) in the hope that the resulting text is somehow easier to compress. In particular, can we transform text so that there will likely be many long runs? (At first glance this may sound impossible, but as we will see, the Burrows-Wheeler transform (BWT) will do just that.) Can we transform text so that characters have very uneven frequencies and hence Huffman should perform well? (Again, this may seem impossible, but can be done.)

The original bzip2 has multiple steps; we will not give the full details of all of them but list here only the ones that are the most interesting from a data structures point of view. Figure 10.5 gives an overview of all the steps, and each of the subsections below discusses each briefly (in reverse order).

#### 10.4.1 Huffman encoding

The last step of bzip2 is a Huffman encoding. Its input is a text that (as we will see) has lots of ‘special’ characters $A'$ and $B'$, and the rest of the source consists of numbers (in the range $\{0, \ldots, 127\}$) that are usually quite small. As such, the distribution of characters is quite uneven and Huffman should perform quite well.
10.4.2 Modified run-length encoding

Assume we are given a text $T_2$ that is known to have long runs of zeroes. We know that if a text has many long runs, then RLE should perform well. Therefore bzip2 (as next-to-last step) performs RLE with a few modifications:

- We only encode runs of zeroes, because it is not very likely that there are many other long runs, and this way we can use fewer bits to encode runs.

- Because we only encode runs of 0s, the alphabet of the output string will be quite big anyway. As such, we can afford to add two more characters $A'$ and $B'$ to the alphabet, and to use these to encode the length of the run.\(^3\) With that, the boundary of the encoding of each run-length is obvious: it ends whenever a “normal” character is encountered. As such, there is no need to use Elias-gamma-codes, which saves almost half of the length of the encoding of the run-length.

- We can use a slightly different encoding called binary bijective numeration, which works as follows:

\[
\begin{array}{cccccccc}
  k & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

(You have encountered this kind of enumeration if you have ever used a spreadsheet: The columns are enumerated the same way, except that they use 26 characters rather than 2.) This uses $\lceil \log(k + 1) \rceil$ characters, rather than $\lceil \log k \rceil + 1$ characters that are used by regular binary encoding. Therefore it is never worse, and usually 1 bit better.\(^4\)

Note that if the input to this was a text $T_2$ that had lots of runs of zeroes, then the resulting text $T_3$ therefore should have lots of $A'$ and $B'$, making it suitable for Huffman encoding as desired. Also note that for any characters other than 0, $A'$ and $B'$, the frequency is the same in $T_2$ and $T_3$, so an uneven distribution of frequencies in $T_2$ will be “passed on” to $T_3$.

10.4.3 Move-to-front transform

Recall that we had the MTF-heuristic when storing a dictionary as an unsorted array. This meant that search is potentially slow, but if we have reasons to believe that we frequently search for some items repeatedly, then the MTF-heuristic means a very fast search.

\(^3\)One of the characters $A'$ and $B'$ can actually be 0, since we will remove all 0s from $T_2$, but we write $A'$ and $B'$ here for ease of understanding how the encoding of the run-length works.

\(^4\)Binary bijective numeration could also be used to make RLE a tiny bit better. This is left as an exercise.
In bzip2, we receive (as result of the first step) a text $T_1$ that has long runs of characters. We can view $T_1$ as a sequence of requests for characters in a dictionary that contains ASCII. If we store the dictionary as an unsorted array $L$ with the MTF-heuristic, we can transform $T_1$ into a sequence of integers by writing, for every such “request-character” $c$, the index where $c$ was found.

Let us illustrate this on an example of text $T_1 = \text{GOOD}$, and for ease of description we use a shortened dictionary $L$ that only stores the characters $\{G, O, D\}$ (but initially in ASCII order). Then text $T_1$ is transformed into $T_2 = \{1, 2, 0, 2\}$ as follows:

The reader may notice that there are lots of 0s, but not actually any long runs of zeroes. That happens because the chosen string is English, and there are very few words in English that repeat characters more than twice. In generally, if $T_1$ has a character repeating $k$ times, then $T_2$ will have a run of $k - 1$ zeroes; this is why for English text the MTF-transform is not particular helpful, but for the output of the Burrows-Wheeler transform it will be very helpful.

The output $T_2$ should also have the property that small indices are much more likely than large indices, presuming we start with ASCII. This holds because the used characters are brought to the front with the MTF heuristic, and so if they are ever used again they should have a much smaller number. Therefore the distribution of characters in $T_2$ (where “character” now means “number in $\{0, \ldots, 127\}$) is quite uneven, making it suitable for Huffman-compression. As noted above, the modified RLE-encoding that we do next preserves this property, except for character 0.

The MTF transform is another example of a text compression/transform that uses an adaptive dictionary, since the dictionary changes with every step. However, the change happens in a well-defined manner that only depends on the currently encoded character. Therefore the encoder can easily emulate this, and no special tricks are needed for decoding. Algorithm 6 and 7 shows the corresponding algorithms. Note that the run-time is proportional to the total time that it takes to find the characters in the dictionary; this is $O(|\Sigma_S| \cdot |S|)$ in the worst-case, but should be much better in practice since frequently used characters should be found quickly.

### 10.4.4 Burrows-Wheeler Transform

Now we turn to the ingredient of bzip2 that makes all the others work: How to transform a text $T_0$ into a text $T_1$ that has lots of runs of characters? We will do this in such a way that $T_1$ is a permutation of $T_0$, i.e., it has the exact same set and frequencies of characters, but the order is different. It would be easy to generate a permutation that has lots of runs of characters (e.g., we could simply sort the characters of $T_0$), but the other crucial
**Algorithm 6: MTF transform encoding**

**Input:** Stream $S$ of characters in alphabet $\Sigma_S$

$L \leftarrow$ array with $\Sigma_S$ in some pre-agreed, fixed order (typically ASCII)

initialize output stream $C$

**while** $S$ has more characters **do**

$c \leftarrow S$.pop()

for $i = 0, 1, \ldots$ **do**

$\text{if } L[i] = c \text{ then break}$

$C$.append($i$)

for $j = i - 1$ **down to 0** **do**

swap $L[j]$ and $L[j + 1]$

**end while**

---

**Algorithm 7: MTF transform decoding**

**Input:** Stream $C$ of indices in $\{0, \ldots, |\Sigma_S| - 1\}$

$L \leftarrow$ array with $\Sigma_S$ in some pre-agreed, fixed order (typically ASCII)

initialize output stream $S$

**while** $C$ has more indices **do**

$i \leftarrow C$.pop()

$S$.append($L[i]$)

for $j = i - 1$ **down to 0** **do**

swap $L[j]$ and $L[j + 1]$

**end while**

---

Requirement is that we must be lossless, i.e., give the permutation, we must be able to recover the original. It is not at all obvious that this can be done. For example, simply sorting the characters is not lossless; the words good and odog would give the same sorted list of characters.

To define how this can be done, we need the concept of a cyclic shift: Given a string $S$ (say as an array $S[0..n-1]$), the $i$th cyclic shift of $S$ is the string $S[i..n-1] \cdot S[0..i-1]$, i.e., the $i$th suffix of $S$, with the “rest” of $S$ appended behind it. For example, for the string $S = \text{alfalphaeatsalfalpha}$ the cyclic shifts are shown on the left side of Figure 10.6.

---

**Burrows-Wheeler encoding**

The Burrows-Wheeler encoding is now very easily obtained: Take the $n$ cyclic shifts of source text $S$, sort them alphabetically, and then output the last character of the sorted
cyclic shifts. Figure 10.6 illustrates this; the resulting encoded string $C$ is hence

$\text{a a f f $ f u e u l l l a a t a}$

There are three things that we must consider:

• Why is this useful for $\text{bzip2}$? Recall that the objective was to obtain a string where there are lots of long runs of characters. In our example, there are indeed such runs; in particular we have a run of three $\text{a}$, and a run of three $\text{f}$. Is this a coincidence?

Recall that typically texts have lots of repeated substrings; our example had substring $\text{alf}$ occur three times. This means that there are 3 cyclic shifts of $S$ that begin with $\text{lf}$ and end with $\text{a}$. So these cyclic shifts begin in the same way, and should end up near each other in the sorted order of cyclic shifts. Note that if they end up being consecutive (as they are in our example) then this results in three consecutive $\text{a}$s in the output, hence a run of characters. It is not guaranteed that repeated substrings lead to repeated characters in the output, because there could be some other cyclic shift that begins with $\text{lf}$ and ends with something other than $\text{a}$. But it is quite likely.

• How do we compute this encoded string efficiently? It is clear that this can be done in polynomial time: we could explicitly write down the cyclic shifts, sort them (MSD radix sort seems best), and then extract the last characters. The problem is that there are $n^2$ characters in the $n$ cyclic shifts together, so this would take time $\Theta(n^2)$, which is too slow for practical purposes. We will discuss below how to do this faster.

• Is this really lossless? It does not seem obvious at all that we can recover the original string from the encoded string. It turns out to be true (and even quite easy to do, though understanding the correctness is not so easy); see the next subsection.
Algorithm 8: Burrows-Wheeler encoding

Input: String $S$
initialize output stream $C$
$\pi \leftarrow$ inverse sorting permutation of cyclic shifts of $S$
for $i = 0 \ldots |S| - 1$ do
  if $\pi[i] = 0$ then
    $C$.append($\$)$
  else
    $C$.append($S[\pi[i] - 1]$)

Efficient BWT encoding

There are a few ingredients that can be used to make BWT encoding more efficient in practice. First, while it is convenient for humans to draw an $n \times n$-matrix as in Figure 10.6, there is really no need for a computer to waste $\Theta(n^2)$ time to do so: The $k$th character of the $i$th cyclic shift is $S[i + k \mod n]$, so we can read any cyclic shift directly from $S$ whenever needed.

Next, we said to sort the cyclic shifts, but actually for extracting the encoding, it suffices to describe what the sorting would have been, or put differently, to give the sorting permutation. It turns out that it is better to have the inverse sorting permutation $\pi$ that satisfies the following: If $s_0, s_1, \ldots, s_{n-1}$ is the sorted order of cyclic shifts, then $s_i$ is the $(\pi(i))$th cyclic shift, i.e., $s_i = S[\pi(i), \pi(i)+1, \ldots, n-1, 0, 1, \ldots, \pi(i)-1]$. In consequence, the $i$th character of the encoding is $S[\pi(i)-1]$ (or $S[n-1] = \$ if $\pi(i) = 0$). Algorithm 8 shows how to extract the Burrows-Wheeler encoding.

It remains to discuss how to find the inverse sorting permutation $\pi$ efficiently. For this we need an observation:

Claim 3. If a string $S$ ends with end-of-word character $\$$, and if $\$ is smaller than all other characters in $\Sigma_S$, then the inverse sorting permutation of the cyclic shifts of $S$ is the same as the inverse sorting permutation of the suffixes of $S$.

Proof. Let us first reformulate the claim in a way that is perhaps easier to understand. We would like to argue that it does not matter whether we sort the cyclic shifts or whether we sort the suffixes instead. This holds if and only if, for any $i \neq j$, we have

$$S[i..n-1] <_{\text{lex}} S[j..n-1] \iff S[i..n-1] \cdot S[0..i-1] <_{\text{lex}} S[j..n-1] \cdot S[0..j-1].$$

To see that this holds, recall that $S[n-1] = \$$, and this end-of-word character does not appear anywhere else in $S$ and is smaller than all other characters. So whenever we lexicographically compare two suffixes or cyclic shifts of $S$, we never need to compare
beyond the place where $ occurs, because at the latest then any tie between the two strings is broken. Therefore, it utterly does not matter what the two strings contain (if anything) after the character $. Since the $th suffix and the $th cyclic shift are identical up to character $, therefore their sorted order is the same.

So now we have reduced the problem of finding $ to the problem of finding the inverse sorting permutation of the suffixes. But for this, we have tools! In particular, if we have the suffix tree, then (presuming children are ordered in ASCII-order) traversing it will visit the suffixes in lexicographic order. So we can obtain $ in linear time by computing the suffix tree of $ and traversing the leaves in order. Since each leaf of the suffix tree knows the index of the suffix that it represents, listing the indices in order gives $.

Slightly slower, but significantly easier to implement, is to use the suffix array, rather than the suffix tree. This can be computed in $O(n \log n)$ time easily (and even the linear-time computation is simpler than the one for suffix trees). The resulting suffix array $A_n$ is exactly the inverse sorting permutation of the suffixes; we can use it directly to extract the Burrows-Wheeler encoding using Algorithm 8.

Burrows-Wheeler Decoding

We now turn to the least obvious ingredient of the Burrows-Wheeler transform: It is possible to recover the input-string from the gobbledigokc that it created. We will illustrate this in an example first, and then give the algorithm. So let us try to recover the source string $ from encoding $ = a n n b $ a a. For this purpose, let us create the matrix $M$ of cyclic shifts. (As we will see later, it is not actually necessary to create this matrix, everything can be done in an array, but having it will be helpful for understanding why the algorithm works correctly.)

• We know that the rightmost column of $M$ is $C$, because that is how the Burrows-Wheeler encoding is created. See Figure 10.7(a).

• We can also quite easily re-create the leftmost column of $M$: We know that this contains all the leftmost characters of all the cyclic shifts, and they are in alphabetic order (since the rows are sorted lexicographically). But the $i$th cyclic shift starts with $S[i]$, so the leftmost column of $M$ contains exactly all characters of $S$. And we also know that $C$ is a permutation of $S$. So we can obtain the leftmost column of $M$ by putting the characters of $C$ in sorted order. See Figure 10.7(b).

• Now we can actually figure out the character to the right of $. Namely, study row 4. Here, the rightmost character is $ and the leftmost character is $. Since these are cyclic shifts, we know that in $ character $ is followed by $. Since $ is the last character of $, hence $[0] = b$. In a similar manner, studying row 4, we can figure out that $[1] = a$. See Figure 10.7(c).
• Now, however, we are seemingly stuck. There are three rows where the rightmost character is an a—which of those should we use to figure out $S[2]$?

• To break this impasse, let us disambiguate the characters in the rightmost column by attaching their row-index, see Figure 10.7(d). Also, for $i = 0, \ldots, n-1$, let $w_i$ be the string of the first $n-1$ characters in row $i$, as illustrated in Figure 10.7(e). For example, we have $w_0 = $ba\ldots, where we don’t know the last three characters yet.

• Consider the three strings $w_0, w_5$ and $w_6$ that are adjacent to the three copies of a. Bringing a to the front, we get three other cyclic shifts of $S$: aw$_0$, aw$_5$ and aw$_6$. But we know that the a in front of $w_i$ has row-index $i$, so these are actually $(a,0)w_0$, $(a,5)w_5$ and $(a,6)w_6$. See Figure 10.7(f).

• Here is the crucial insight: We know that $w_0 \lessdot w_5 \lessdot w_6$, because the three cyclic shifts $w_0a, w_5a$ and $w_6a$ appear in this order among the lexicographically sorted cyclic shifts. Therefore $aw_0 \lessdot aw_5 \lessdot aw_6$. But recall that the a in front of $w_i$ is $(a,i)$, so we have

$(a,0)w_0 \lessdot (a,5)w_5 \lessdot (a,6)w_6$.

In particular this tells us exactly which a on the left side corresponds to which a on the right side: they appear in the same order. The same holds for all other characters,
so we now know exactly the correspondence of characters between the rightmost and leftmost column. See Figure 10.7(g).

- Now we can easily figure out the entire string, because we can read (starting at $) which character follows which. Specifically

$\rightarrow (b, 3) \rightarrow (a, 6) \rightarrow (n, 2) \rightarrow (a, 5) \rightarrow (n, 1) \rightarrow (a, 0) \rightarrow ($

so the entire string decodes to **banana**$.$

Burrows-Wheeler decoding is one of those algorithms where the algorithm itself is actually very simple; what made the above steps so complicated is arguing that it works correctly. Here are the steps of the decoding summarized (and the pseudocode is in Algorithm 9):

- Disambiguate the characters in $C$ by their index.
- Sort the characters of $C$ (stably!) to get array $A$.
- Find the position of the end-of-word character $\$$ in $C$.
- Go from the current character of $C$ to the character in $A$ at the same index.
- Since this character in $A$ knows its index in $C$, we can repeat the process until we again reach character $\$$, and we have then read $S$ in order.

We can sort the characters using count-sort; this takes $O(n + |\Sigma_C|)$ time. Everything else takes $O(n)$ time, so the run-time for Burrows-Wheeler decoding is $O(n + |\Sigma_C|)$.

Let us see one more example where we do the Burrows-Wheeler decoding directly with this algorithm. Let $C =$ ard$rcaaaabb$. After disambiguation, we have

which after sorting becomes

We find $\$$ at index 3 in $C$, look up that $A[3]$ is (a,7), look up that $A[7]$ is (b,11), and so on until we have the entire string **abacabacabra**$$. (It does feel like magic, doesn’t it?)

$(\$, 3) $\rightarrow (a, 7) \rightarrow (b, 11) \rightarrow (r, 4) \rightarrow (a, 8) \rightarrow (c, 5) \rightarrow (a, 9) \rightarrow (d, 2) \rightarrow (a, 6) \rightarrow (b, 10) \rightarrow (r, 1) \rightarrow (a, 0) \rightarrow (\$, 3)$
**Algorithm 9:** Burrows-Wheeler decoding

**Input:** Encoded string $C$ as array $C[0..n-1]$

$A \leftarrow$ array of size $n$

for $i = 0$ to $n-1$ do

$A[i] \leftarrow (C[i], i)$

Stably sort $A$ by first entry

for $j = 0$ to $n$ do  

if $C[j] = \$ \text{ then break}$

initialize output stream $S$

repeat

$j \leftarrow$ second entry of $A[j]$

$S.append(C[j])$

until $C[j] = \$

---

### 10.4.5 bzip2 discussion

$bzip2$ is reported to compress in practice much better than LZW encoding (though there are variants of LZW that compress even better). The price to pay is that $bzip2$ is quite slow. The run-length encoding takes time $O(n)$ (where we use as $n$ here the maximum length among the texts $T_0, T_1, T_2, T_3$). The Huffman-encoding takes time $O(n + |\Sigma| \log |\Sigma|)$, which is close to linear. We give the analysis here in terms of a general alphabet $\Sigma$ that includes all alphabets used; normally $\Sigma \approx \text{ASCII}$ has constant size, so an additive term depending on $|\Sigma|$ is acceptable, but a multiplicative term might be prohibitive. This is why the MTF-transform could be considered to be quite slow: it could take time $O(n|\Sigma|)$ in the worst-case since the lookup for each character can take $\Theta(|\Sigma|)$ time. Luckily enough this should be rare. The real bottle-neck for time is the Burrows-Wheeler transform, which takes time $O(n \log n)$ if done via suffix arrays obtained via modified MSD radix sorting. (There are ways to find the suffix array in linear time, or one could build a suffix tree, but both of those are slower than modified MSD radix sorting in practice.)

Interestingly enough, decode is much faster in $bzip2$ than encoding: Both the MTF-transform and the Burrows-Wheeler-transform can be decoded in linear time without need to anything more complicated than countsort.

One drawback of $bzip2$ is that it needs the entire source text at once, because to encode we must look at various characters of the cyclic shifts to sort them. Thus, in contrast to previous schemes, it is not possible to implement $bzip2$ such that both input and output are streams, making it unsuitable for text that does not fit into memory.

Summarizing, while $bzip2$ compresses very well, it has some issues, and research in compression techniques is ongoing.