

CMPS340 File Processing
Ziv-Lempel-Welch Compression and Decompression

Compression Algorithm:

Input: We assume that the object *input* provides access to a string (i.e., sequence of characters) over the alphabet $A = \{a_1, \dots, a_m\}$. Specifically, repeated calls to its *getChar()* method allow us to scan the string from beginning to end, one character at a time. The method returns a special value, \perp , when we attempt to read past the end of the string.

Output: An encoding of the input string as a sequence of nonnegative integers (encoded as bit strings) is written via the *output* object's *emit()* method.

Notes: The *emit()* method has two arguments, the first of which is the nonnegative integer to be emitted, the second of which is the number of bits that should be used to encode that integer.

The dictionary, d , is modeled as a function of type $\mathcal{N} \rightarrow A^*$ (i.e., mapping nonnegative integers to strings over the alphabet A), which is to say that it is a set of ordered pairs (k, y) , where $k \in \mathcal{N}$ and $y \in A^*$. (By design of the algorithm, d will necessarily be *injective* (i.e., *one-to-one*), which means that it has an inverse function, d^{-1} .) The meaning of $(k, y) \in d$ (or, equivalently, $d(k) = y$, or $d^{-1}(y) = k$) is that string y has been assigned the numeric code k . By $d.range()$ we mean the set of strings to which numeric codes have been assigned, i.e., $\{y \in A^* \mid \text{for some } k, (k, y) \in d\}$.

```
// initialize dictionary so that length-one strings corresponding to
// members of the source alphabet are each assigned a numeric code.
d := { }; // set d to an empty dictionary;
do for each i in 1..m
    d.insert(i, a_i); // inserts the ordered pair (i, a_i) into d
od;
n := m + 1; // n holds the number of entries in d, plus 1
x := λ; // set x to empty string
c := input.getChar();
do while c ≠ ⊥
    if x · c ∈ d.range() then
        x := x · c;
    else
        output.emit(d-1(x), ⌈lg n⌉);
        d.insert(n, x · c); // inserts the ordered pair (n, x · c) into d
        n := n + 1;
        x := c; // x becomes string of length one
    fi;
    c := input.getChar();
od;
if x ≠ λ then
    output.emit(d-1(x), ⌈lg n⌉);
fi;
output.emit(0, ⌈lg(n + 1)⌉); // emit numeric code for ⊥
```

Decompression Algorithm:

Input: We assume that the object *input* provides access to a bit string that was produced by the compression algorithm detailed above. (Hence, this bit string encodes a sequence of nonnegative integers, which, in turn, encodes some string over the alphabet $A = \{a_1, \dots, a_m\}$.) Specifically, the method *input.getNum()* consumes the number of bits specified by its argument and yields the nonnegative integer represented by that bunch of bits. Consistent with the compression algorithm above, end of data (i.e., \perp) is signaled by zero.

Output: Via the *output* object's *emit()* method, the algorithm writes that string over A that was encoded by the input bit string.

Notes: The dictionary, d , is as described above. The method *firstChar()*, when applied to a (nonempty) string, yields its first character. The \cdot operator denotes string concatenation, so that, for example, $z \cdot \text{firstChar}(\text{nextZ})$ is the string formed by concatenating the first character of *nextZ* to the end of z .

```
// initialize dictionary so that length-one strings corresponding to
// members of the source alphabet are each assigned a numeric code.
d := { }; // set d to an empty dictionary;
do for each i in 1..m
    d.insert(i, a_i); // inserts the ordered pair (i, a_i) into d
od;
n := m + 1; // n holds the number of entries in d, plus 1
k := input.getNum(⌈lg n⌉);
if k = 0 then
    z := λ;
else
    z := d(k);
    k := input.getNum(⌈lg(n + 1)⌉);
fi
do while k ≠ 0
    output.emit(z);
    if k = n then
        nextZ := z · firstChar(z);
    else
        nextZ := d(k);
    fi;
    d.insert(n, z · firstChar(nextZ));
    n := n + 1;
    z := nextZ;
    k := input.getNum(⌈lg(n + 1)⌉);
od;
output.emit(z);
```