Experiment No. 4

Aim: To study LZW coding.

Equipment/Software: Python codes

Theory:

This is a popular variant of LZ78, developed by Terry Welch in 1984. Its main feature is eliminating the second field of a token. An LZW token consists of just a pointer to the dictionary. To best understand LZW, we will temporarily forget that the dictionary is a tree, and will think of it as an array of variable-size strings. The LZW method starts by initializing the dictionary to all the symbols in the alphabet. In the common case of 8-bit symbols, the first 256 entries of the dictionary (entries 0 through 255) are occupied before any data is input. Because the dictionary is initialized, the next input character will always be found in the dictionary. This is why an LZW token can consist of just a pointer and does not have to contain a character code as in LZ77 and LZ78.
The principle of LZW is that the encoder inputs symbols one by one and accumulates them

in a string I. After each symbol is input and is concatenated to I, the dictionary is searched for string I. As long as I is found in the dictionary, the process continues. At a certain point, adding the next symbol x causes the search to fail; string I is in the dictionary but string Ix (symbol x concatenated to I) is not. At this point the encoder (1) outputs the dictionary pointer that points to string I, (2) saves string Ix (which is now called a *phrase*) in the next available dictionary entry, and (3) initializes string I to symbol x.
To illustrate this process, we again use the text string *si sid Eastman easily teases sea sick seals*. The steps are as follows:
1. Initialize entries 0–255 of the dictionary to all 256 8-bit bytes. The first symbol s is input and is

found in the dictionary (in entry 115, since this is the ASCII code of s). The next symbol i is input, but si is not found in the dictionary. The encoder performs the following: (1) outputs 115, (2) saves string si in the next available dictionary entry (entry 256), and (3) initializes I to the symbol i.
2. The r of sir is input, but string ir is not in the dictionary. The encoder (1) outputs 105 (the ASCII code of i), (2) saves string ir in the next available dictionary entry (entry 257), and (3) initializes I to the symbol r.

To understand how the LZW decoder works, we recall the three steps the encoder performs each time it writes something on the output stream. They are (1) it outputs the dictionary pointer that points to string I, (2) it saves string Ix in the next available entry of the dictionary, and (3) it initializes string I to symbol x. The decoder starts with the first entries of its dictionary initialized to all the symbols of the alphabet (normally 256 symbols). It then reads its input stream (which consists of pointers to the dictionary) and uses each pointer to retrieve uncompressed symbols from its dictionary and write them on its output stream. It also builds its dictionary in the same way as the encoder.

CODE:

```
def compress(uncompressed):
    """Compress a string to a list of output symbols."""

    # Build the dictionary.
    dict_size = 256
    dictionary = dict((chr(i), i) for i in range(dict_size))
    # in Python 3: dictionary = {chr(i): i for i in range(dict_size)}

    w = ""
    result = []
    for c in uncompressed:
        wc = w + c
        if wc in dictionary:
            w = wc
        else:
            result.append(dictionary[w])
            # Add wc to the dictionary.
            dictionary[wc] = dict_size
            dict_size += 1
            w = c

    # Output the code for w.
    if w:
        result.append(dictionary[w])
    return result

def decompress(compressed):
    """Decompress a list of output ks to a string."""
    from io import StringIO

    # Build the dictionary.
    dict_size = 256
    dictionary = dict((i, chr(i)) for i in range(dict_size))
```

```
    # in Python 3: dictionary = {i: chr(i) for i in range(dict_size)}

    # use StringIO, otherwise this becomes O(N^2)
    # due to string concatenation in a loop
    result = StringIO()
    w = chr(compressed.pop(0))
    result.write(w)
    for k in compressed:
        if k in dictionary:
            entry = dictionary[k]
        elif k == dict_size:
            entry = w + w[0]
        else:
            raise ValueError('Bad compressed k: %s' % k)
        result.write(entry)

        # Add w+entry[0] to the dictionary.
        dictionary[dict_size] = w + entry[0]
        dict_size += 1

        w = entry
    return result.getvalue()
```

OUTPUT: