AD1 – Algorithms and Data Structures I (course 1DL210)

Assignment 4: Data Compression

Due by 23:59:59 on Friday 16 May, 2008

Introduction

The purpose of *data compression* is to take an input file A and, within a reasonable amount of time, transform it into an output file B in such a way that B is smaller than A and that it is possible to reconstruct A from B. A program that converts A into B is called a *compressor*, and one that undoes this operation is called a *decompressor*. Programs such as *gzip* perform this function. Compression enables us to store data more efficiently on storage devices or transmit data faster using communication facilities, since fewer bits are needed to represent the actual data.

A compressor cannot guarantee that B will always be smaller than A. Indeed, if this were possible, then what would happen if one just kept compressing the output of the compressor?! Any compressor that succeeds in compressing some files must thus also actually fail to compress some other files. Nevertheless, compressors tend to work pretty well on the kinds of files (especially those generated by computer novices using Micro$oft Office) that are typically found on computers, and they are widely used in practice, especially for pictures, movies, and sounds.

The Ziv-Lempel Algorithm

We here consider a version of the *Ziv-Lempel data compression algorithm*, which is the basis for most popular compression programs, such as *WinZip*, *zip*, and *gzip*. You may find this algorithm a little difficult to understand at first, but your program could be quite short.

It is an example of an *adaptive* data compression algorithm: the code used to represent a particular sequence of bytes in the input file may be different for distinct input files, and may even be different if the same sequence appears in more than one place in the input file.

Compressor

The Ziv-Lempel compressor maps strings of input characters to numeric codes. To begin with, each character of the set of characters, called the *alphabet*, that may occur in the text file is assigned a code. For example, suppose the input file starts with the string:

aaabbbbbbaabaaba

This string is composed of the characters **a** and **b**. Assuming the alphabet is just $\{\mathbf{a}, \mathbf{b}\}$, initially **a** is assigned the code 0 and **b** the code 1. The mapping between character strings and their codes is kept in a dictionary. Each dictionary entry has two fields: a *code* and a *string*. The character string represented by the field *code* is stored in the field *string*. The initial dictionary for our example is given by the first two columns below:

code	0	1	2	3	4	5	6	7
string	**a**	**b**	**aa**	**aab**	**bb**	**bbb**	**bbba**	**aaba**

Beginning with the dictionary initialised as above, the Ziv-Lempel compressor repeatedly finds the longest prefix p of the unprocessed part of the input file that is in the dictionary and outputs its code. Furthermore, if there is a next character c in the input file, then pc (denoting the string p followed by the character c) is assigned the next available code and inserted into the dictionary. This strategy is called the *Ziv-Lempel rule*.

Example 1 Consider the example string **aaabbbbbbaabaaba** above. The longest prefix of the input that is in the initial dictionary is **a**. Its code 0 is output and the string **aa** (for $p = \mathbf{a}$ and $c = \mathbf{a}$) is assigned the code 2 and entered into the dictionary. Now, **aa** is the longest prefix of the remaining string that is in the dictionary. Its code 2 is output and the string **aab** (for $p = \mathbf{aa}$ and $c = \mathbf{b}$) is assigned the code 3 and entered into the dictionary. Even though **aab** has the code 3 assigned to it, the code 2 for **aa** is actually output! The suffix **b** will be a prefix of the string corresponding to the next output code. The reason for not outputting 3 is that the dictionary is *not* part of the compressed file. Instead, the dictionary has to be reconstructed during decompression using the compressed file. This reconstruction is possible only if we adhere strictly to the Ziv-Lempel rule: see the next subsection. Following the output of the code 2, the code 1 for **b** is output and **bb** is assigned the code 4 and entered into the dictionary. Then, the code 4 for **bb** is output and **bbb** is entered into the dictionary with code 5. Next, the code 5 is output and **bbba** is entered into the dictionary with code 6. Then, the code 3 is output for **aab** and **aaba** is entered into the dictionary with code 7. Finally, the code 7 is output for the entire remaining string **aaba**. The example string is thus encoded as the sequence 0214537 of codes, and the final dictionary is as given above.

Uncompressor

For decompression, we read the codes one at a time and replace them by the strings they denote. The dictionary can be dynamically reconstructed as follows. The codes assigned for single-character strings are entered as (*code*, *string*) pairs into the dictionary at the initialisation (just as for compression). This time, however, the dictionary is searched for an entry with a given code (rather than with a given string). The first code in the compressed file necessarily corresponds to a single character and so may be replaced by that character, which is already in the dictionary. For all other codes x in the compressed file, we have two cases to consider:

1. If the code x is already in the dictionary, then the corresponding string, denoted by $\mathit{string}(x)$, is extracted from the dictionary and output. Furthermore, we know that for the code q that precedes x in the compressed file the compressor created a new code for the string $\mathit{string}(q)$ followed by the first character of $\mathit{string}(x)$, denoted by $\mathit{fc}(x)$. So we also enter the pair (next code, $\mathit{string}(q)\mathit{fc}(x)$) into the dictionary.

2. If the code x is not yet in the dictionary, then the uncompressed text segment corresponding to the compressed file segment qx has the form $\mathit{string}(q)\mathit{string}(q)\mathit{fc}(q)$, where q is the code that precedes x in the compressed file. Indeed, during compression, the string $\mathit{string}(q)\mathit{fc}(q)$ was assigned the new code x in the dictionary and the code x was output. So we output $\mathit{string}(q)\mathit{fc}(q)$ and enter the pair $(x, \mathit{string}(q)\mathit{fc}(q))$ into the dictionary.

Example 2 Consider the example string **aaabbbbbbaabaaba**, which was compressed in Example 1 into the code sequence 0214537. The dictionary is initialised with the pairs $(0, \mathbf{a})$ and $(1, \mathbf{b})$. The first code is 0, so its string **a** is output. The next code, 2, is still undefined. Since the previous code 0 has $\mathit{string}(0) = \mathbf{a}$ and $\mathit{fc}(0) = \mathbf{a}$, we have $\mathit{string}(2) = \mathit{string}(0)\mathit{fc}(0) = \mathbf{aa}$, so **aa** is output and $(2, \mathbf{aa})$ is entered into the dictionary. The next code, 1, triggers output **b** and $(3, \mathit{string}(2)\mathit{fc}(1)) = (3, \mathbf{aab})$ is entered into the dictionary. The next code, 4, is not yet in the dictionary. The preceding code is 1, so $\mathit{string}(4) = \mathit{string}(1)\mathit{fc}(1) = \mathbf{bb}$. The pair $(4, \mathbf{bb})$ is entered into the dictionary and **bb** is output. Similarly for the next code, 5, where $(5, \mathbf{bbb})$ is entered into the dictionary and **bbb** is output. The next code is 3, which is already in the dictionary, so $\mathit{string}(3) = \mathbf{aab}$ is output and the pair $(6, \mathit{string}(5)\mathit{fc}(3)) = (6, \mathbf{bbba})$ is entered into the dictionary. Finally, when the code 7 is read, the pair $(7, \mathit{string}(3)\mathit{fc}(3)) = (7, \mathbf{aaba})$ is entered into the dictionary and **aaba** is output. The original example string has thus been reconstructed, and the final dictionary is again as given on the previous page.

Implementing the Ziv-Lempel Algorithm

The data structure to be used by the *compressor* is a *hash table* storing integer codes for string keys. New codes are entered into the hash table for given strings, and the hash table is queried with strings for the corresponding codes, if any. For this assignment, we limit ourselves to the codes 0 through 4095. The ASCII codes 0 through 255 will be used for single-character strings, even though the character with ASCII code 0 will never be encountered in the input file. So the first new code that the compressor actually assigns will be 256. If more than 4096 codes are needed, then do not generate new codes but use the available ones; this may give less compression, but otherwise is not a problem.

The *decompressor* can actually be simpler. Since we just query on integer codes, rather than on strings, use an *array* of 4096 strings and initialise it for the first 256 single-character strings. For instance, array position 65 (which corresponds to ASCII symbol **A**) *must* contain the string "**A**". Starting with position 256, new strings are dynamically entered into this array.

Work To Be Done

Implement the following functions:

- `compress` compresses a file, given as a list of characters. You may use a suitable hash-table implementation of the Moscow ML Library (see `http://www.itu.dk/people/sestoft/mosmllib/`). To avoid a number of problems, make this function return the sequence of codes as a list of integers. Actual compression would involve rather advanced SML details, such as binary input/output, bit packing, etc, which are really beyond the scope of this assignment. There may thus not be any actual compression here in terms of bytes consumed.

- `decompress` decompresses a list of integer codes into a list of characters. You may use a suitable array implementation of the Moscow ML Library.

Give, in comments within the program, your explicit reasoning establishing the worst-case runtime complexities of these functions.

If both functions are correct, then `decompress(compress cs) = cs` for any character list `cs`.

Example 3 The character list is

$$[\mathbf{a}, \mathbf{a}, \mathbf{a}, \mathbf{b}, \mathbf{b}, \mathbf{b}, \mathbf{b}, \mathbf{b}, \mathbf{b}, \mathbf{a}, \mathbf{a}, \mathbf{b}, \mathbf{a}, \mathbf{a}, \mathbf{b}, \mathbf{a}]$$

if and only if the code list is

$$[97, 256, 98, 258, 259, 257, 261]$$

once we adjust the codes of Examples 1 and 2 as discussed on top of this page.

Grading

Your solution is graded in the following way:

- If your program was submitted before the deadline turns hard, loads under Moscow ML version 2.01, and is a serious attempt at implementing and commenting (under at least the coding convention) all the requested functions, then you get 30 points (before any penalty deductions for being late compared to the soft deadline); otherwise, you get 0 points.
- We will run three tests on your programs:
  - We will decompress with *your* `decompress` function a code list obtained with *our* `compress` function. If our input and your output are identical, then you get 20 points.
  - We will decompress with *our* `decompress` function two code lists obtained with *your* `compress` function. For each identical input/output pair, you get 15 points.

  We reserve the right to run these tests automatically, so be careful with names and argument orders.
- Your program is graded on style and comments (including specifications, representation conventions and invariants, and recursion variants), provided it does not fail on all the tests we perform. This covers 10 points.
- Your complexity analysis is graded for correctness of results and explicitness of reasoning. This covers 10 points.

Have fun!