Program Design & Data Structures (course 1DL201)
Uppsala University – Autumn 2012 / Spring 2013
Homework 3: Data Compression

Prepared by Pierre Flener

Lab: Wednesday 13 February 2013
Submission Deadline: 18:00:00 on Friday 15 February 2013
Lesson: Friday 22 February 2013
Resubmission Deadline: 18:00:00 on Thursday 28 February 2013

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 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 *gzip*, *zip*, and *winzip*. 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.

Decompressor

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 all possible 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).

Given a code x of the compressed file, we denote by $string(x)$ the corresponding segment of the decompressed text. Also, given a string s, we denote its first character by $fc(s)$.

The first code in the compressed file necessarily corresponds to a single character of the alphabet, which is already in the dictionary, and so that character is output.

For each other code x in the compressed file, let us assume that it follows code q, which necessarily has already been decompressed into $string(q)$. This means that code sequence qx in the compressed file corresponds to the concatenation $string(q)string(x)$ in the decompressed file. The pair $(q, string(q))$ is already in the dictionary, while $string(x)$ is what we want to compute. We have two cases to consider, depending on whether pair $(x, string(x))$ is also already in the dictionary or not:

- If the pair $(x, string(x))$ is already in the dictionary, then we output $string(x)$. Furthermore, we have to apply the Ziv-Lempel rule in order to augment the dictionary, the way it

was done by the compressor when outputting code q. So we enter the pair (next available code, $string(q)fc(string(x))$) into the dictionary.

- If the code x is not yet in the dictionary, then we must be in the situation where the compressor, after having generated a new code x (by the Ziv-Lempel rule) upon outputting code q for $string(q)$, has immediately used x to encode the next text segment, namely $string(x)$. This means that $string(x)$ must be of the form $string(q)c$, with c being the character following $string(q)$ in the decompressed file. But since x has been immediately used, c is indeed the first character of $string(x)$, which in turn is $fc(string(q))$. Hence, we can infer that $string(x) = string(q)fc(string(q))$. So we output $string(q)fc(string(q))$ and put the pair $(x, string(q)fc(string(q)))$ 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 $string(0) = \mathbf{a}$ and $fc(string(0)) = \mathbf{a}$, we have $string(2) = string(0)fc(string(0)) = \mathbf{aa}$, so **aa** is output and $(2, \mathbf{aa})$ is entered into the dictionary. The next code, 1, triggers output **b** and $(3, string(2)fc(string(1))) = (3, \mathbf{aab})$ is entered into the dictionary. The next code, 4, is not yet in the dictionary. The preceding code is 1, so $string(4) = string(1)fc(string(1)) = \mathbf{bb}$. 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, 3, is already in the dictionary, so $string(3) = \mathbf{aab}$ is output and the pair $(6, string(5)fc(string(3))) = (6, \mathbf{bbba})$ is entered into the dictionary. Finally, when the code 7 is read, the pair $(7, string(3)fc(string(3))) = (7, \mathbf{aaba})$ is entered into the dictionary and **aaba** is output. The original example string has 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 4048. 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 4049 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 a ***vector*** of 4049 strings and initialise it for the first 256 single-character strings. For instance, vector position 65 (which corresponds to ASCII symbol **A**) ***must*** contain the string "**A**". Starting with position 256, new strings are dynamically entered into this vector.

Work To Be Done

Implement the following functions in a file called `zivLempel.sml`, making sure that they pass the training test cases (at `http://www.it.uu.se/edu/course/homepage/pkd/ht12/assignments/assignment3-training.sml`), which are ***not*** be included:

- `compress` compresses a file, given as a list of characters. Use the Poly/ML hash-table implementation described at `http://www.it.uu.se/edu/course/homepage/pkd/ht12/`

`assignments/assignment3-hashing.txt`. 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 between 0 and 4048 inclusive into a list of characters. Use the vector implementation of the Standard ML Basis Library (at `http://www.standardml.org/Basis/vector.html`).

If both functions are correct, then `decompress (compress cs) = cs`, for any character list `cs`. **Hint:** For converting a character into a string, do not use `Char.toString` (which behaves strangely with special characters) but rather use `String.str` (or just `str`, as the `String` library is opened automatically).

In a separate report in PDF format, give an explicit reasoning establishing the worst-case runtime complexities of these functions.

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 above.

Grading

Your solution is graded with 0 to 100 points in the following way:

1. If your solution was submitted by the deadline, your program loads under Poly/ML version 5.5.0 (Uppsala modified 1) and passes ***all*** the training test cases, and your solution is deemed to be a serious attempt at implementing, commenting (under at least the coding convention), ***and*** analysing ***all*** the requested functions, then you get 20 points; otherwise (including when no solution was submitted by the submission deadline), you get a U grade for this homework (even if an insufficiently commented program is actually correct). **Hint:** Test your submitted code in a freshly started ML interpreter in order to make sure your code does not work because of some old data or functions lying around in the interpreter.

2. Your program is run on an unspecified number of orthogonal grading test cases that satisfy all pre-conditions but ***also*** check boundary conditions. Each test case is a decompression with ***your*** `decompress` function of a code list obtained with ***our*** `compress` function, or vice-versa. ***We reserve the right to fail a correct program that has a quite suboptimal time complexity and thus takes an unreasonable amount of time on very large test cases.*** For each fully correct test result, you get a suitable amount of points, the total being 40 points. We reserve the right to run these tests automatically, so be careful to match exactly the imposed file names, function names, and argument orders.

3. Your program is graded for style and comments (including function specifications, datatype representation conventions and invariants, as well as recursion variants), provided it does not fail on all the grading test cases. This covers 20 points.

4. Your complexity analysis is graded for correctness of results and explicitness of reasoning. This covers the remaining 20 points.

We assume that by submitting a solution you are certifying that it is solely the work of your group, except where explicitly attributed otherwise. We reserve the right to use plagiarism detection tools and point out that they are extremely powerful.