Project 3 – Compression

# Due Date: 2nd Dec

Overview: In this project, you will be creating code to compress and decompress files.

Goal: To gain experience with compression algorithms.

Requirements: You are going to implement a compression algorithm in Python 3.x. You have been given a set of example files. The files in the folder called “simple” use a limited subset of characters: lowercase letters, spaces, newlines, apostrophes, dashes and periods. These ones are probably the easiest ones to start with initially. The files in the “text” subfolder use arbitrary characters from the Unicode encoding – your code will earn more points on the rubric if it can handle an arbitrary alphabet. You should add further test files as appropriate.

Your program should offer a simple menu interface of options:

* Your algorithm should be able to encode the contents of a file into a sequence of bits and then write the bit-sequence into a separate file. Provide the user with a menu of all the files available for encoding and allow the user to specify a filename for writing the encoded data to – you can use a folder structure to keep the encoded files separate from the non-encoded ones.
* Write an algorithm for decompressing a compressed file back into uncompressed form. Depending on how you’ve structured your algorithm, you may be using the same tree, regardless of file being decompressed, or a different tree for each file. If the latter, you are free to set up the menu structure so that the user can only decompress the most recently compressed file.  
  If you use different trees, you can earn one bonus rubric point, by writing your encoding tree into the file with your bit-string such that you can then read it back in and use it to decompress the file without having to store it in memory.

Your code should compute the following statistics each time the encode function is called on a file:

* Runtime in milliseconds of the encoding process.
  + If your tree varies from file to file, then the time taken to generate the tree should be included in the runtime. Similarly, if your encoding process uses character frequencies that are computed from the specific file, then you need to include the time for that computation in your runtime.
* Number of bits needed by your algorithm to encode the contents of the file.
* Number of bits needed by a fixed length encoding to encode the contents of the file.
* The compression ratio achieved.

For the decoding process, you only need to track the runtime in milliseconds.

Algorithm: In addition to your code, you need to turn in a well-formatted document containing the pseudo-code for your core algorithm (written in the same format as the textbook uses for pseudo-code) along with any proofs you decide to include about your algorithm – see grading section below.

Grading: Your algorithm will be graded according to the rubric below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** |
| **Compression** | Can compress plain text files from “simple” folder. | Compressed size at least as good as Huffman **or** can compress any text file. | Compressed size at least as good as Huffman **and** can compress any text file. | Compressed size better than benchmark code. | Best compression ratio of all student submissions. |
| **Decompression** | Inaccurate decompression. |  | Decompresses file, but some mistakes in decompressed version. |  | Recreates file exactly as was before compression. |
| **Time** | Code always terminates in finite time. | Runtime of code no worse than clean implementation of Huffman. | Best time performance of all student submissions. |  |  |
| **Efficiency Analysis** | Defines input size and basic operation for domain. States without proving what efficiency class the algorithm belongs to. | Defines input size and basic operation for domain. Proves that algorithm belongs to Θ(g(n)). |  |  |  |

Your project will receive a final score based on the number of points from the above rubric that it accrues. The table below summarizes the minimum number of points from the rubric that would be necessary to earn various grades. Extra credit is possible.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Grade:** | 50 | 47.5 | 45 | 42.5 | 40 | 37.5 | 35 |
| **Points on Rubric:** | 13 | 12 | 11 | 10 | 9 | 8 | 7 |

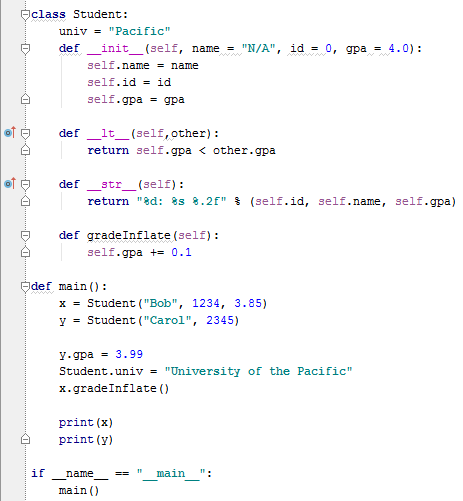
Runtime Guides: The following benchmarks can be used in determining whether your code is matching the performance of a clean Huffman implementation and the benchmark algorithm. The benchmark times were based on running the code on one of the lab computers in Anderson 105. The times are averages over 100 runs.

|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **Huffman** | | **Benchmark** |
| **Compressed Size (in number of bits)** | **Runtime (in msecs)** | **Compressed Size (in number of bits)** |
| simple/data1.txt | 18 | 1.370 | 8 |
| simple/data2.txt | 1162 | 2.250 | 608 |
| simple/data3.txt | 13365 | 6.350 | 9297 |
| simple/data4.txt | 577289 | 162.9 | 462259 |
| text/data1.txt | 1263 | 2.030 | 633 |
| text/data2.txt | 15607 | 6.600 | 9830 |
| text/data3.txt | 30573 | 10.39 | 18715 |
| text/data4.txt | 689859 | 174.6 | 500822 |
| text/data5.txt | 3187 | 3.030 | 3187 |

Submission Instructions: Please submit a .zip or .7z file containing:

* the code for your project,
* the data files you tested on
* your document containing the algorithm pseudo-code and any proofs.

Helpful Syntax: You may want to create a class for some part of the code that you are writing in this project. Since we didn’t specifically cover how to write classes in Python, I am including the following example code as a reference. Further details can be found at any of the python documentation webpages.



Member variables are created whenever you use self.*varname* in a member function, e.g. in the 3 lines of the \_\_init\_\_ function. Variables that are declared outside of a member function, like univ, are shared between all the copies of an object (like a static variable in C++). So both of the Student objects, x and y, will share one semi-global copy of the univ variable and any change that one of them makes will be seen by the other one.

\_\_init\_\_ is the constructor for the class. Like all member functions, its first parameter is the object itself (called self). The following three parameters are used to pass in starting values, which init then copies into the corresponding member variables. One thing to watch out for, when using member variables, you must always use self.*varname*; if you forget the “self” then Python will create a separate local variable that happens to have the same name.

Notice that the parameters for \_\_init\_\_ have been given default values (with the = symbol), which means that they are optional. The user can omit one or more of those parameter values (from right to left) as we did in main when declaring object y.

The function gradeInflate is an example of a regular member function. It can manipulate its own name, id and gpa member variables by modifying self.varname. It can modify the shared univ variable by using Student.univ.

In main you can see that member functions are called using the dot notation familiar from C++. Notice that member variables are not protected and can be modified from main as well.

The remaining two member functions, \_\_str\_\_ and \_\_lt\_\_, are overloaded implementations of standard functions, which is why their names start and end with two underscores. \_\_str\_\_ is called whenever you call “print” or “str” on an object and the job of \_\_str\_\_ is to create a string representation of the data inside the object. \_\_lt\_\_ is used to tell whether one object is less than the other. It is useful if you want to create a heap of objects because heapq uses the less than function to determine where to place items in a min-heap.