Home Assignment 3
Due Date: 30.5.19

In this task you will implement Huffman code in order to compress files. This task relates to the OOP, working with files, and data structures. Before starting you should be familiar with the Huffman code. There are a lot of sources online, but you can start here:
https://en.wikipedia.org/wiki/Huffman_coding
In the Wikipedia page you'll see near the bottom some variations on the basic algorithm. They are **beyond** our scope; you should implement only the basic binary algorithm.

As in the previous homework we provide you with an interface. Your job is to write a class which inherits this interface and implements the required methods. You are free to implement as many methods as you like beyond the required ones.

Constructor:

The constructor receives a single argument "input_file_path". Within the constructor do whatever you want (initialize properties, call other methods, etc.). By calling the constructor the program will go to the file located in the specified path, read its contents, and write a compressed version of the file in the same directory as the source file.

Other required methods:

decompress_file – The method receives a single argument "input_file_path". This is a path to a compressed file. The program will go to the file located in the specified path, read its contents, and write a decompressed version of the file in the same directory as the source file. It returns the path to the decompressed file as string
calculate_entropy – This method calculates the entropy associated with the distribution of symbols in a previously compressed file. Though we believe the concept of entropy is new to you, (you'll learn more about it in future courses), the formula for computation is simple, and appears both in the Huffman Coding wiki page and here. The function takes no arguments (but has accesses to all class properties) and returns the calculated entropy (float).

Some more details:

- The required symbol size is one byte, i.e. regardless of whether you're compressing an binary or a text file, your code should take one byte of data at a time, and encode it into the corresponding codeword.
- As you'll learn by reading the provided references, each byte will be encoded into a codeword of variable length.
- Compressed file size – The Huffman coding ensures us that the compressed file size (in bytes) should be:
$$\frac{n}{8}H(X) \leq file\ size \leq \frac{n}{8}(H(X)+1)$$
where $H(X)$ is the entropy, and n is the number of symbols (bytes) in the original file. Thus, finding the entropy will tell you if your compression is good enough.
- In the formula for entropy the base of the log has to be determined. The above formula, and our grading code assumes the base of 2.
- As you will see building a Huffman code involves a binary tree, some references regarding trees are found here, but there are much more:
  - https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm
  - https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion-and-without-stack/
- You will need to work with files, if you have questions look here.

Compression and file types:

- We require from you to implement compression for both, binary and text files, i.e. our grading system will ask you to compress both either files with extension: '.txt' or '.bin'.
- This usage is the same for the user, i.e. you test which extension was given and act accordingly.
- Regardless of the original file extension, the compressed file is a binary one.
- The distinction is important since we require you to return an exact replica of the original file upon decompressing, this includes the extension of course.

Successes Criteria:

- The grading system will ask your code to compress several files (txt, and bin).
- For each file your code will have to compress the file (upon construction), and then decompress it (when we call the decompress method), such that the decompressed file is an exact replica of the original (to the bit).
- Furthermore, we will test the generation of the compressed file, and we shall look that it shrunk sufficiently in size.
- Lastly, we shall check your implementation of the the entropy calculation.

Restrictions:

- We allow you to import only three modules:
    - OS – to handle directories
    - Math
    - Total ordering from functools ("from functools import total_ordering")
- The last one is not must, and you can solve the problem without it, it saved us some time.
- The allowed import statements appear already in the student_id_template.py. Do not change them! Adding other modules will fail you.
- You may wish to define more classes (look at recitation 7) which are instantiated from within the HuffmanCoding class. This is fine. However, since you cannot import them, you must put all of your classes in the same file.
- Runtime – No reason why your code should take more than a minute to run for the whole test (this of course depends on file size). The one-minute limit regards running the supplied test files.

Tips:

- Since this work involves several parts which are conceptually separate (working with files, building the tree, etc.) it may be easier to build separate classes for separate tasks. Then the main HuffmanCoding class will hold instances of sub classes. This is a matter of taste, do it only I you find it convenient. If you do this remember the restriction about import.
- You don't want to write files to nonexistent directories… Write the files in the same directory where you found the original.

Important:

- When you'll reading about the Huffman code, you will find that: Sometimes the number of bits you need to write to the files isn't a multiple of 8 (remember that there are 8 bits in one byte). In other words, the compressed version of the data doesn't occupy an exact number of bytes. This poses a problem: **The operating system requires every file to occupy fully a certain number of bytes. You cannot write 2.5 bytes to the hardisk.** Possible solution:
    - Add zeros at the end of the last partial byte
    - To be able to decode correctly, you need to know how many zeros you added. You can do this by deciding that you add another whole byte at the beginning of the file. This byte will hold a number with the number of redundant zeros you added in the end.
- One mistake will rule all – Unlike previous HW assignments, in this one it is impossible to check each component of the work separately. For instance, if you fail to write a file, we cannot check your compression. **Make sure your code passes the supplied test file to avoid unpleasantries.**

Files and submission:

- **An interface was uploaded to Moodle.** You should be familiar with it and use it
- while implementing your solution. You can start coding with our **template** file that was uploaded to Moodle.
- Your code will be uploaded using an **import statement** into our auto-grading code. For debugging purposes, you should use the statement if __name__ == '__main__'.
- Document your code - for yourself, and for good practice. Your comments should be written in English only.
- You should submit a single .py file through the Moodle website with the name of YOUR_ID_NUMBER.py were YOUR_ID_NUMBER should be your id number.
- The run time is limited to 1 minute for supplied test files.
- Avoid copying!