Institut for Matematik og Datalogi
Syddansk Universitet, Odense

28. April 2025

Project: Part III

DSK814

This project consists of three parts: each part has its own deadline, so that the work is spread over the entire semester. The deadline for part III is *Friday, May 30th at 11:59 p.m.* The three parts I/II/III are not equal in size, but are approximately 15/25/60 in size.

The project must be completed in groups of two or three.

Goal

For project part III you will create your own tool to compress files via the Huffman encoding. You will implement two programs: one to encode/compress a file and one to decode it.

Make sure you understand the Huffman algorithm before reading the task description, see (Cormen et al., 4. Edition: Section 15.3 until midway of page 43 [Cormen et al., 3. Edition: Section 16.3 until page 433]).

Task 1

In short: you will implement a python program that reads a file and returns a Huffman encoded version of it, see below for an exact definition of the input and output. The program should be named `Encode.py`, and must be able to run like this:

```
python Encode.py nameOfOriginalFile nameOfCompressedFile
```

The input file should be seen as a sequence of bytes (8 bits) where each byte represents a character. Therefore, there are $2^8 = 256$ possible different

characters in the input, and our alphabet has size 256. Throughout we refer to characters as bytes. In Python, the call `read(1)` from file objects will read one byte from a file if the file is opened in binary reading mode (with the argument `'rb'` in `open`).

Python 3 has a type called "byte objects", which you can think of as (immutable) lists of integers with values between 0 and 255.[1] Calling `read(1)` from file objects returns a byte object of length one. When you have reached the end of the file, `read(1)` returns a byte object of length zero (instead of length one). Consequently, a byte is eight bits in files on disk, but arrives in your Python program as a byte object of length one. If you access the first (and only) element of this byte object, you get an integer between 0 and 255. If the byte object is named `b`, the first element is accessed as `b[0]`.

You need this integer to represent the bytes along the way in your Python program.

As for the binary number system, refer to the table below for the relationship between `int`'s and bytes. When opening files in binary mode, the functions `read()` and `write()` implicitly use this relationship, so you don't have to deal with it in the Python program as long as you represent bytes by integers between 0 and 255.

Integer	Byte
0	00000000
1	00000001
2	00000010
3	00000011
⋮	⋮
254	11111110
255	11111111

The `Encode` program should work like this:

1. Scan the input file and create a table (a list with 256 entries) of the frequency of the individual bytes (remember that bytes are integers between 0 and 255 and can be used as indexes in lists).

[1] Slightly confusingly, an entire byte object in Python is represented using ASCII characters if possible. For example, a byte object `s` with content `[120,121,122]` is represented as `b'xyz'` (with `b` for "binary"), where the character `x` in the ASCII table has number 120, but individual elements in byte objects *are* integer (e.g. is `s[0]` equal to `120`).

2. Run the Huffman algorithm with the table from item 1 as input (all 256 entries, including those with frequency zero[2]).

3. Convert Huffman tree to a table (a list of 256 entries) of code words for each of the possible bytes (remember that bytes are represented as integers between 0 and 255, and can be used as indexes in lists).

4. Write the 256 frequencies to the output file (i.e. the 256 integers in the table from item 1).

5. Scan the input file again. Along the way, find the code word for each byte (by looking in the table of code words from item 3), and write the bits of this code word to the output file.

Scanning the input file twice is preferable to doing it once and saving its contents in the program for further use, as this increases the program's RAM consumption from $O(1)$ to the size of the input file (which can be very large).

The implementation of item 3 should perform a recursive traversal of the Huffman tree (similar to an inorder traversal of a search tree), thereby generating all code words. During the traversal, maintain the code word corresponding to the path from the root to the current node. As you go down the tree, the password must be expanded by either 0 or 1. As soon as you reach a leaf, save the code word in the table. All code words (which are not of equal length) must be represented by a string of characters `'0'` and `'1'`. After a table lookup, each string can be traversed character by character and converted to bits that are written to the output file.

Use the implementation of `PQHeap.py` from Part I to implement the Huffman encoding via a priority queues (see Cormen et al., 4. Edition: page 434 [Cormen et al., 3. Edition: page 431]). Instead of using a list of numbers for the implementation of priority queue as in Part I, use a list of `Element`'s in Part III (the code for `Element` is provided with this project). An `element` has two fields `key` and `data`, where `data` represents a tree (see description below), and `key` is the overall frequency of the tree (i.e. a number). When comparison operators (`==`, `<`, `=<`, etc.) are used between `Element`'s, the implementation will automatically compare the `key`'s of the two `Element`'s.

Therefore, you can simply use the code from `PQHeap.py`. The only two changes are: when using `insert(A,e)`, then `e` is an `Element` and using

[2]Huffman's algorithm works only for alphabets with at least two characters. If you omit characters with frequency zero, files with content of type `aaaaaaa` would result in an alphabet of size one. Hence, this requirement

`extractMin(A)` returns an `Element` (namely, the one with the smallest `key` among all `Element`'s in the priority queue). Furthermore, we provide a program `PQSortElements.py`, which is a variant of `PQSort.py` from Part I, just adjusted to sort `Element`'s instead of integers. Note that you need your own `PQHeap.py` from Part I to run the program. Refer to `PQSortElements.py` in order to learn how

- To create `Element`'s,
- To access the fields in an `Element` with name `e` as `e.key` and `e.data`,
- `Element`'s are inserted with `PQHeap.insert(pq,e)`,
- To extract `Element`'s with `PQHeap.extractMin(pq)`.

In the implementation of Huffman's algorithm, use binary trees with one byte (i.e. an integer with value between 0 and 255) stored in the leaves, see the figure in Cormen et al., 4. Edition: page 435 [Cormen et al., 3. Edition: page 432]. However, do *not* store the frequency in the nodes (as illustrated in the book) as only the frequency of the root is used in Huffman's algorithm, which is already stored as `key` in the `Element` containing the tree. Thus, inner nodes are empty! It's up to you whether you create two different Python classes for inner nodes and leaves, or use the same Python class for all nodes (then use -1 for inner nodes in the byte field to distinguish them from leaves).

The exact specification of the output of the `Encode` program is: write each of the 256 integer as 32 bits (taking up $256 \cdot 32$ bits in total), which specifies the frequencies of the 256 possible bytes in the input, then give the bits of the Huffman encoding of the input.

Note that for short files (or long files that cannot be compressed significantly with the Huffman method), the output of `Encode` may be slightly longer than the original file, as we use some space to store the frequency table. However, avoiding or limiting this situation is not part of the project.

When writing a code word as output, write one bit at a time. There are no methods in Python to read and write single bits to disk (the smallest unit is a byte), but you can use the library `bitIO.py` that contains the classes `BitReader` and `BitWriter` which provide such methods. There are also methods to read and write integers as 32 bits, which should be used when writing the frequency table to the output. See the provided program `text.py` for examples of how to use the methods.

Note that the output from `Encode` is *not* readable with normal editors as the saved Huffman code words only make sense when interpreted as the Huffman codes from *your* particular Huffman tree. Other programs don't know them.[3] Only your `Decode` method from task 2 will be able to read that part of the output. The same issue applies to the first part of the output, which has stored the frequency table with `writeint32bits(intvalue)` from the class `BitWriter`. This part can only be read by `readint32bits()` from the class `BitReader`.

> SUMMARY: In task 1, call `read(1)` from file objects to read bytes from the input file (the original file). You need to use the methods `writeint32bits(intvalue)` and `writebit(bit)` from the `BitWriter` class in the `bitIO.py` library to write integer (for frequency table) and bits (for Huffmans codes) to the output file (the compressed file). Both files must be opened in "binary mode". When a `BitWriter` is instantiated, it must have a file object as an argument.

Task 2

You need to implement a Python program that reads a file with data generated by your program from task 1 and writes to the original (uncompressed) content to a file. The program should be named `Decode.py` and run like this:

```
python Decode.py nameOfCompressedFile nameOfDecodedFile
```

Using only one scan of the input, the `Decode` program should work like this:

1. Load the table of frequencies for the 256 bytes from the input file.
2. Generate the same Huffman tree as the program from task 1 (i.e. use the *same* implementation in both programs such that in situations where the Huffman algorithm has multiple options the same option is chosen).

[3] Other programs will try to interpret the bits as they normally would, such as utf8, latin1 or ASCII-encoded text which results in meaningless output.

3. Use this Huffman tree to decode the rest of the bits in the input file, while writing the original version of the file as output. This is done by using the read bits to navigate down through the tree (while they are being read). When a leaf is reached, its byte is printed, and continues from the root.

Note that the total number of bits from the printout of the Huffman codes of the supplied library are rounded up to a multiple of eight, i.e. to a whole number of bytes. If necessary zero bits are added at the end when the file is closed, because computers can only save files that contain an integer number of bytes. These (possibly) added bits must not be decoded, as extra bytes may appear in the output. Hence, find the total number of bytes in the original file by summing up the frequencies during the decoding, and keep track of how many bytes you have written during the reconstruction of the uncompressed file.

To write bytes to a file, use `write(bytes([b]))` from file objects and the built-in function `bytes()` to write a byte represented by the integer `b`. The file object in question must be opened in binary writing mode (with the argument `'wb'` in `open`).

> SUMMARY: In task 2, use the methods `readint32bits()` and `readbit()` from the class `BitReader` in library `bitIO.py` to read integers (for the frequency table) and bits (for the Huffmans codes) from the compressed input file. Use `write(bytes([b]))` (where `write()` is from file objects and `bytes()` is a built-in function) to write bytes to the recreated original output file. Here, `b` is an integer representing the byte to be written. Both files must be opened in "binary mode". When a `BitReader` is instantiated, it must have a file object as argument.

Formalities

You only need to submit your Python source files. Make sure that your code is well commented. It must contain the names and SDU logins of the group members. Your programs will be tested with many types of files (`txt`, `doc`, `jpg`,...), and you should do this yourself before submission, but you do not have to document these tests.

The file must be submitted electronically in `Digital Eksamen`. The files `Encode.py` and `Decode.py`, as well as all other files that necessary to run them like `PQHeap.py` must be submitted either as individual files or as one zip-archive (with all files at the top level, i.e. without any directory structure).

During the submission, you must declare the group by stating the names of all members. You only need to submit once per group. Please note that during the submission, you can create temporary "drafts", but you can only submit once.

Submit by:

Friday, May 30th at 11:59 p.m.

Please note that submitting someone else's code or text, whether copied from fellow students, from the internet, or in other ways, is exam cheating, and will be treated very seriously according to current SDU rules. You will also not learn anything. In short: only people whose names are mentioned in the submitted file may have contributed to the code.