URGENT ─ FEATURE ENHANCEMENT REQUIRED

The *New Software Application* (NSA) company asks for a little improvement to the steganography tools you just delivered: the tools must become able to compress data before injecting it into the image.

Command-line interface

In particular, a new argument should be added to both tools, **`-compress`**, with no parameters. The intended behavior when the flag is activated is:

- the `dissimulate` tool will first compress the message, then store it on the image, suffixed by the magic number;
- the `reveal` tool will obviously first read the compressed message then decompress it.

The Huffman Coding

The NSA asks you to use *Huffman Coding* as compression algorithm. The technique employs a binary tree to assign *codes* to *symbols*; the tree structure is based on the frequency of appearance of the symbols on the message, so the length of the resulting codes is inversely proportional to the frequency of the original symbol, compressing thus the message.

Compression

An example message: `aabdecdaebade`, 13 characters, 13 bytes in ASCII encoding.
The frequency for each symbol can be easily obtained iterating over the string:
`a: 4, b: 2, d: 3, e: 3, c: 1`.

Then, the tree can be constructed, starting from the leaves: a leaf node is created for each symbol, containing the symbol itself and the frequency. These nodes are then put into a queue, ordered by ascending frequency:
`{c:1, b:2, d:3, e:3, a:4}`

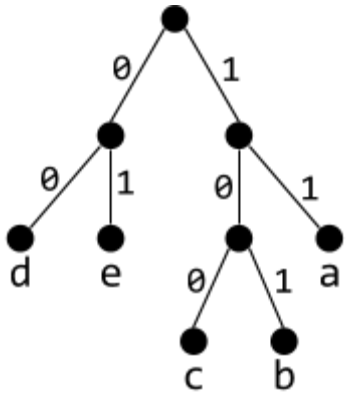
Here starts the iterative part of the algorithm; for each iteration:

- remove the two nodes at the beginning of the queue (i.e. less frequent);
- create an internal node with these two nodes as children, and with as frequency the sum of the frequencies;
- add this node to the queue in the right place, so to keep the order.

Iterations:

- ➢ `{c:1, b:2, d:3, e:3, a:4}`
  take `c` and `b`, create the node `(c,b)` with frequency `1+2=3`, reinsert it into the queue, keeping the order
- ➢ `{d:3, e:3, (c,b):3, a:4}`
- ➢ `{(c,b):3, a:4, (d,e):6}`
  take `(c,b)` and `a`, create the node `((c,b),a)` with frequency `3+4=7`, ...
- ➢ `{(d,e):6, ((c,b),a):7}`
- ➢ `{((d,e),((c,b),a)):13}`

When only a node is present on the queue, that is the root of the Huffman tree:



Each symbol is then replaced by a code of a variable length, obtainable by reading the tree from the root to the corresponding leaf node, bit '0' representing the left child and bit '1' the right one. The dictionary resulting from the example is:
`d: 00, e: 01, c: 100, b: 101, a: 11`

The Huffman code is a "prefix code"; this means that a code is never prefix for another one; for this reason, the compressed message does not need separators between the codes, as their recognition is unmistakable.

Compressing the message symbol by symbol, these are the resulting bits:
`11 11 101 00 01 100 00 11 01 101 11 00 01`
Packed into bytes:
`11111010 00110000 11011011 10001`

Decompression

Knowing the compressed message and the dictionary, it is possible to decode the original message.

To decompress, consume bit-by-bit the message, saving the sequence in an intermediate buffer; for each bit added to the buffer, check if there is a matching code; if it is the case, output the corresponding symbol, empty the buffer and continue.

Dictionary: `d: 00, e: 01, c: 100, b: 101, a: 11`
Bits of compressed message: `11111010001100001101101110001`
Initialize variables: `buffer: empty, output: empty`
Iterations:

- `buffer: 1` no matching code
- `buffer: 11` a is matching ⇒ `buffer: empty, output: a`
- `buffer: 1` no matching code
- `buffer: 11` a is matching ⇒ `buffer: empty, output: aa`
- `buffer: 1` no matching code
- `buffer: 10` no matching code
- `buffer: 101` b is matching ⇒ `buffer: empty, output: aab`
- `buffer: 0` no matching code
- ...

Implementation details

The original message will be compressed byte-by-byte, the expected output to be dissimulated inside the image must be in this format:

- The first byte: the number of symbols - 1 (i.e. if the message was "`abbacb`": `3-1=2`), unsigned.
- For each symbol:
  - one byte for the symbol itself;
  - one byte for the code length, unsigned;
  - one or more bytes containing the bits of the code corresponding to the symbol, zero-padded[1].
- One byte that specify how many bits of the last byte are used (zero meaning that the compressed message is byte-aligned), unsigned.

An acceptable output for the message `aabdecdaebade` is:

```
00000100  // number of symbols - 1
01100100 00000010 00000000  // ASCII for d, two, code for d, padding
01100101 00000010 01000000  // ASCII for e, two, code for e, padding
01100011 00000011 10000000  // ASCII for c, three, code for c, padding
01100010 00000011 10100000  // ASCII for b, three, code for b, padding
01100001 00000010 11000000  // ASCII for a, two, code for a, padding
00000101  // five: number of bits used in the last bit
11111010 00110000 11011011 10001000  // compressed message, padding
```

That is 21 bytes. The space saved is:

$$(1 - \frac{compressed\ size}{original\ size}) \cdot 100 = (1 - \frac{21}{13}) \cdot 100 = -61.5\%$$

In this case the compression was clearly not a wise choice, as the compressed message is longer than the original, a negative saving confirm this scenario.

Acceptance criteria

- Avoid regressions: previous already-working functionalities must not be damaged.
- The standard interface specified in the details above ensure the compatibility between the products of different teams.
- The NSA gives you some advices on the degree of importance of a number of different message formats to be compressed; these levels of compliance are:
  - L1: message containing a single (repeated) character;
  - L2: message containing two (repeated) characters, a bit per symbol is enough;
  - L3: message encoded with a predefined dictionary (lowercase alphabet[2]);
  - L4: detection of negative saving, and inform the user on stdout;
  - L5: arbitrary message.
- Both tools should expose an extra `-show` argument, that if used with `-compress`, will print on stdout the dictionary (as *hex: code*) and the compressed message (as *bits*) of the compression or input of the decompression.

---

[1] In order to align a content to a specified data structure (bytes, in this case), some spare non-significant bits are added. This technique is called *padding*.

[2] https://en.wikipedia.org/wiki/Letter_frequency#Relative_frequencies_of_letters_in_the_English_language

The expected output for the example is:

```
0x64: 00
0x65: 01
0x63: 100
...
11111010001...
```

- Improve the `-metrics` parameter on `dissimulate` tool, by adding `compression_savings`, in percentage, and `compression_time`, in milliseconds.
- The solution proposed should support extensibility and evolution, as the company could ask for cyphering in the upcoming months.

Report

You are requested to write a report, illustrating these points:

- Difficulties you encountered during the development: present them and explain your solutions.
- Extensibility: how is it easy to add a new compression algorithm, if needed? What about chaining different tools (such as compression and encryption)?
- Tests: illustrate your methods and state the coverage.
- Differences between procedural and object-oriented programming: do your previous statements still hold? Give some strength and weak points for each paradigm.
- Benchmarks: how does your implementation perform with respect to message length and dictionary length? Present your testbed and your results.
- Complexity: calculate the complexity of the compression and decompression algorithms, and describe your reasoning.
- Your project and software engineering: strength and weak points.

Deliveries

- The rules of the previous part still hold: a GIT repository with 2 directories and the two `build.sh` scripts.
- A demo is requested by NSA for Thursday morning.
- Final products and report on Sunday at 23:59. The expected tag is “vFinal”.

END OF DOCUMENT.