Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

uOttawa

L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Electrical Engineering
and Computer Science

Programming Assignment 1 (15%) CSI2110/CSI2510

Due : November 6, 11:59PM, 2021 (suggestion : complete by Oct 27)

Late assignment policy: ***1min-24hs late are accepted with 30% off; no assignments accepted after 24hs late.***

Huffman Code for File Compression

Problem Description

Lossless compression is a type of data compression that allows the original data to be recovered from the compressed data. Huffman encoding is a compression technique that reduces the number of bits needed to store a message based on the idea that more frequent letters should have a shorter bit representation and less frequent ones should have a longer bit representation. Text compression is useful when we wish to reduce bandwidth for digital communications, so as to minimize the time required to transmit a text. Text compression is also used to store large documents more efficiently, for instance allowing your hard drive to contain as many documents as possible.
In this assignment, you will implement the Huffman encoding (compression) and decoding (decompression) algorithms.
This assignment will use **binary trees** to store the Huffman tree and it will use **priority queues** as part of the greedy algorithm that builds the Huffman tree.

1 How the Huffman code works

Suppose we have 4 possible letters that appear on a message, a string or a file: A, C, T, G. We could encode these letters with a fixed length code using 2 bits for each letter:

letter	A	C	G	T
fixed length encoding	00	01	10	11

Using a fixed length encoding, the message AACGTAAATAATGAAC which has 16 letters can be encoded with 32 bits as 00000110110000001100001110000001

Huffman encoding would instead check how many times each letter appears in the text and choose a variable length code depending on letter frequencies:

letter	A	C	G	T
Frequency in text	9	2	2	3
Huffman code	1	010	011	00

With Huffman encoding the same message can be encoded with 27 bits (saving 15% space)

110100110011100110001111010

Now imagine a more drastic scenario where a text message uses a standard 8-byte encoding for each character (like in UTF-8 character encoding) able to support 256 different characters. Suppose that our specific message is the same as in the example above (AACGTAAATAATGAAC), so that for the 4 characters we have the same frequencies as shown in the table above but the other 252 other characters have frequency 0. Now the message above would be encoded in UTF-8 using 16*8=128 bits or 16 bytes. Now comparing our Huffman encoding above that has 27 bits, which is less than 4 bytes we can now save 75% space.

In another scenario, suppose you want to store the entire human genome composed of a sequence of 3 billion nucleotide base pairs. This can be represented by a sequence of 3 billion characters each being one of A (adenine), T (thymine), G (guanine), C (cytosine), which could be stored using 3 billion bytes using UTF-8 encoding which is about 3GB. Huffman encoding for such a file would depend on the relative frequency between these four letters, but assuming about 25% frequency of each of these 4 letters (and of course absence of any other of the 252 character), Huffman would encode each letter with 2 bits yielding a compressed file of 750MB.

How to build the Huffman code from the letters in a text and their frequencies?

The Huffman code is always a prefix code, that is, no codeword is a prefix of any other codeword. This generates no ambiguity. When decoding our example

110100110011100110001111010

we have no doubt that the first word is an A since no other codeword starts with a 1; similarly, we decode the next A. Then the next bits will be read one by one until we conclude unequivocally that the next codeword can only possibly be 010 which is a C. Try continuing this example to see how the original text can be decoded using the Huffman code given in the table in the previous page.

Huffman's algorithm produces an optimal variable-length prefix code for a given string X, using the character frequencies in X. It is based on the construction of a binary tree T that represents the code. Here we quote the textbook by Goodrich et al. 6th edition page 595:

"Each edge in T represents a bit in a codeword, with an edge to the left child representing a 0 and an edge to the right child representing a 1. Each leaf v is associated with a specific character and the codeword for that character is defined by the sequence of bits associated with the edges in the path from the root of T to v. Each leaf v has a frequency, f(v), which is simply the frequency in the string X of the character associated with v. In addition, we give each internal node v in T a frequency, f(v), that is the sum of the frequencies of all the leaves in the subtree rooted at v."

For the previous example, where X=``AACGTAAATAATGAAC'', and the frequencies and codewords given in the previous table, we have the following Huffman tree:

```
            16
          /    \
         7     A(9)
       /   \
    T(3)    4
          /   \
        C(2)  G(2)
```

In order to understand how the tree can be used for decoding, use the tree to decode the encoded string 1101001100111001100011111010 to obtain back X:

Step 1: Bits are read representing going left or right until a leaf with a character is achieved so that a character is decoded.
Step 2: Go back to the root and repeat Step 1, until the file ends.

From root, 1 (go right), decode 'A'
From root, 1 (go right), decode 'A'
From root, 0 (go left), 1 (go right), 0 (go left), decode 'C'
From root, 0 (go left), 1 (go right), 1 (go right), decode 'G'
….
From root, 0 (go left), 1 (go right), 1 (go right), decode 'C'

So we can retrieve from 1101001100111001100011111010 the decoded string

	1	1	010	011	00	1	1	1	00	1	1	00	011	1	1	010
	A	A	C	G	T	A	A	A	T	A	A	T	G	A	A	C

You can find another example in page 596 of the textbook. Inspect that to better understand Huffman trees.

Now, we just need to explain the algorithm to build the Huffman tree: the Huffman coding algorithm begins with each of the characters of the string X being the root node of a single-node binary tree. Each iteration of the algorithm takes **two binary trees with the smallest frequencies and merges them into a single binary tree** creating a root of the new binary tree having the sum of the frequencies of the two subtrees. This process is repeated until only one tree is left.
The construction of the tree in the example above is illustrated next:

```
Start:          A(9), C(2), G(2), T(3)

Iteration 1:    A(9), T(3) ,       4
                                 /   \
                             C(2)   G(2)

Iteration 2:     A(9),        7
                            /    \
                        T(3)      4
                                /   \
                            C(2)   G(2)

Iteration 3:            16
                      /     \
                       7     A(9)
                   /    \
              T(3)      4
                      /   \
                 C(2)   G(2)
```

At each iteration, we need to remove 2 trees with minimum frequency and insert 1 combined tree with the sum of the subtree's frequencies in the list of trees. These two operations indicate we need to use a priority queue to remove each minimum efficiently. The algorithm to build a tree is given next:

```
Algorithm Huffman(X):
    Input: String X of length n with d distinct characters
    Output: Huffman tree for X
    1. Compute the frequency f(c) of each character c of X.
    2. Initialize a priority queue P
    3. For each character c in X do {
    4.      Create a single-node binary tree T storing c
    5.      Insert T with key f(c) into P }
    6. While P.size() > 1 do {
    7.      e1= P.removeMin() with e1  having key f1 and value T1
    8.      e2= P.removeMin() with e2  having key f2 and value T2
    9.      Create a new binary tree T with left subtree T1 and right subtree T2
    10.     Insert T into P with key f1+f2
            }
    12. Entry e= P.removeMin() with e having tree T as its value
    13. return tree T
```

Proving that the Huffman tree algorithm produces an optimal prefix code is beyond the scope of this assignment, but this can be found in other more advanced textbooks. This is an example where the ``greedy method" is successful in finding the optimal solution for a problem.
Using the priority queue P implemented as heap, the above algorithm runs in O(n + d log d) for a string of length n with d distinct characters.

2 Implementation details and what you must do

Huffman encoding (compression)

When using Huffman encoding for compressing a file, the compressed file must include information to reconstruct the Huffman tree necessary in the decoding; that is, we need somehow to inform the character frequencies so that your decoder constructs the tree in exactly the same way as your encoder (breaking ties for equal frequencies in a consistent manner).
In this assignment, this information is stored in an ArrayList<Integer> freqLetters indexed by the possible characters/bytes in the file. Thus freqLetters.get(i) stores the frequency of byte with value i. We extend the possible characters (corresponding to ASCII codes 0..255) to have a special character we encode as being character 256, which represents the end-of-input, say ♣. This is needed because we must be writing bits in the encoding as bytes that will be written to the output file; since the Huffman encoding of a file will not necessarily give a multiple of 8 bits, the last incomplete byte will have to be padded with 0's which could confuse the decoder unless we explicit use a code that indicates "end-of-input" so we know the input ended and ignore the padded zeros.
So, in fact in the previous example with a file containing: AACGTAAATAATGAAC we would encode if at the end we have an end-of-input character ♣, giving the frequencies:

letter	A	C	G	T	♣
Frequency in text	9	2	2	3	1

The Huffman tree generated will be different from the previous example since we have an extra character ♣ with frequency 1.

Since ASCII/UTF-8 code for A, C, G, T are 65, 67, 71, 84, respectively and we reserve code 256 for end-of-input character ♣, the Frequency Table will be as follows:

index	0	1		64	65	66	67	68	69	70	71	72		83	84	85		255	256
freqLetters	0	0	…	0	9	0	2	0	0	0	2	0	…	0	3	0	…	0	1

The steps of the Huffman encoding are given below and corresponding method specified:

1. **Read the input file once and compute letter frequencies**
   `ArrayList<Integer> buildFrequencyTable(InputStream input)`
   Read one byte at a time, calculating the letter/byte frequencies and storing in the Frequency table.

2. **Build the Huffman tree based on the letter frequency**
   `HuffmanTreeNode buildEncodingTree(ArrayList<Integer> freqTable)`
   For this follow the pseudocode for Algorithm Huffman given above.

3. **Compute the Huffman encoding for each letter**
   `ArrayList<String> buildEncodingTable(HuffmanTreeNode encodingTreeRoot)`
   Traverse the tree accumulating a string of bits on the way to each leaf, storing in ArrayList<String> codes, for the character/byte in each leaf the string corresponding to its code in a ArrayList<String> codes. For example, if we find that G is encoded as 011 then index 71 of this array list will contain string "011".

4. **Write the header of the output file which must contain the frequency table.**
   `codedOutput.writeObject(freqTable);`

5. **Read input file a second time, converting each letter/byte into its Huffman encoding and sending these bits to output file.**
   `void encodeData(InputStream input, ArrayList<String> encodingTable, OutputStream output)`
   Read the input file a second time one byte at a time, converting each byte into its Huffman encoding using the ArrayList<String> codes; the corresponding bits are sent to the output. The conversion and accumulation of bits to be sent as bytes to the output will be done with the help of a class BitOutputStream, already programmed inside the Huffman class. At the end send the code of end-of-input special character must be send to output via BitOutputStream. Remember to close BitOutputStream so incomplete bytes are padded with 0s and send to output (see method BitOutputStream.close())

Once these various methods are programmed by you, the encoding function runs as follows:

```
public void encode(String inputFileName, String outputFileName) throws IOException {
        System.out.println("\nEncoding "+inputFileName+ " " + outputFileName);

        // prepare input and output files streams
        FileInputStream input = new FileInputStream(inputFileName);
        FileInputStream copyinput = new FileInputStream(inputFileName);// create copy to read input twice
        FileOutputStream out = new FileOutputStream(outputFileName);
        ObjectOutputStream codedOutput= new ObjectOutputStream(out); // use ObjectOutputStream to print
objects to file

        ArrayList<Integer> freqTable= buildFrequencyTable(input); // build frequencies from input
        HuffmanTreeNode root= buildEncodingTree(freqTable); // build tree using frequencies
        ArrayList<String> codes= buildEncodingTable(root);  // buildcodes for each character in file
        codedOutput.writeObject(freqTable); //write header with frequency table
        encodeData(copyinput,codes,codedOutput); // write the Huffman encoding of each character in file
}
```

Huffman decoding (decompression algorithm)

The steps of the Huffman decoding are:

1. **Read the letter frequencies from the input file header and rebuild the Huffman tree.**
   Note that you already have a method used in encoding that creates the Huffman tree based on frequencies:
   `HuffmanTreeNode buildEncodingTree(ArrayList<Integer> freqTable)`

2. **Read the remaining bits of the input files producing the decoded file.**
   `void decodeData(InputStream input, HuffmanTreeNode encodingTreeRoot, OutputStream output);`
   The bits will be read one by one, and the Huffman tree traversal followed according to the bits until a letter is retrieved and send to the output; then back to the root we repeat, sending the remaining letters to the output. This algorithm is exemplified at the top of page 3. Take care that the end-of-input letter (code 256) is not print to output but is used to stop the decoding. Reading bits from the input will be done with the help of class `InBitStream` already given to you inside `class Huffman`.

Once these various steps are programmed by you, the decoding function runs as follows:

```
public void decode (String inputFileName, String outputFileName) throws IOException,
ClassNotFoundException {
    System.out.println("\nDecoding "+inputFileName+ " " + outputFileName);
    // prepare input and output file streams
    FileInputStream in = new FileInputStream(inputFileName);
    ObjectInputStream codedInput= new ObjectInputStream(in);
    FileOutputStream output = new FileOutputStream(outputFileName);

    ArrayList<Integer> freqTable = (ArrayList<Integer>) codedInput.readObject(); //read header with
frequency table
    HuffmanTreeNode root= buildEncodingTree(freqTable);
    decodeData(codedInput, root, output);
}
```

3 Assignment Tasks and Mark Breakdown (75 marks)

Inspect `class Huffman` to be completed by you.
Three inner classes are already given to you: `HuffmanTreeNode` (node of Huffman tree), `OutBitStream` (auxiliary to output bytes from bits) and `InBitStream` (auxiliary to retrieve bits from read bytes).
Methods `encode` and `decode` are given with the code shown above.
The methods that these methods call, corresponding to the steps of encoding and decoding explained above must be implemented by you.
To program method `buildEncodingTree` you will need a heap priority queue. Feel free to use the HeapPriorityQueue class given in net.datastructures package, or use a heap priority queue produced in Lab 4.

Inspect `class TestCompression`. This is used to test encoding and decoding various files and also let you specify your own tests. It can also perform tests given in a separate file (see *tests.txt* provided)

Your code should output information regarding the tasks being executed and file sizes as follows:

```
Encoding genes.txt genes.huf
Number of bytes in input : 13608
Number of bytes in output: 3578

Decoding genes.huf genesRecovered.txt
Number of bytes in input : 3578
Number of bytes in output: 13608
```

You must handin inside a zip file:

- all classes needed to run your code (*.java)
- readme file (explaining status of code (working, partially working, known bugs, and anything that can help marking)
- file with output of running your file on “tests.txt”: `TestCompression T texts.txt`

Marking Scheme (out of 60 points)

Correctness of solutions provided:	9 points (15%)
Quality of programming:	9 points (15%)
Method `buildFrequencyTable`	6 points (10%)
Method `buildEncodingTree`	15 points (25%)
Method `buildEncodingTable`	9 points (15%)
Method `encodeData`	6 points (10%)
Method `decodeData`	6 points (10%)

Appendix:

Sample run with printouts of Frequency Table and Encoding Table:

```
Encoding genes.txt genes.huf
FrequencyTable is=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 3672, 0, 1836, 0, 0, 0, 2916, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5184, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
EncodingTable is=[null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
10, null, 1101, null, null, null, 111, null, null, null, null, null, null, null, null, null, null, null,
null, 0, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, 1100]
Number of bytes in input : 13608
Number of bytes in output: 3578

Decoding genes.huf genesRecovered.txt
FrequencyTable is=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 3672, 0, 1836, 0, 0, 0, 2916, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5184, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
Number of bytes in input : 3578
Number of bytes in output: 13608
```

As an exercise, reconstruct in paper the Huffman tree using the EncodingTable.
Verify this is the same Huffman tree you obtain using the frequencies in the FrequencyTable.

Note that the Huffman tree and Encodings in different programs may be slightly different depending on which subtrees are placed on the left and on the right by the program (there may be changes of 0's for 1's). This should not affect the correctness of decoding as long as the Huffman tree construction is done by the same algorithm (your encoder and your decoder will be consistent). In addition, this should not change the total encoding size, which is expected to match.

Sample run for string example in this handout AACGTAAATAATGAAC.

From the EncodingTable below we see that **AACGTAAATAATGAAC♣** was encoded by the sequence of bits: `1 1 0111 010 00 111 00 1100 010 1 1 0111 0110` when packed into bytes we get: `11011101 00011100 11000101 10111011 00000000` (in this case we need to pad 7 bits to complete the last byte). Your decoder read these bytes it must ignore the padding, because when you decode 0110 you know it represents the end-of-input character, so no more bits should be looked at. Note that the end-of-input character itself is just for your control and will not be part of the decoded file.

```
Enter command > E ACTGhandout.txt ACTGhandout.huf

Encoding ACTGhandout.txt ACTGhandout.huf
FrequencyTable is=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 9, 0, 2, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
EncodingTable is=[null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, 1,
null, 0111, null, null, null, 010, null, null, null, null, null, null, null, null, null, null, null,
null, 00, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null,
null, null, null, 0110]
Number of bytes in input : 16
Number of bytes in output: 5

Command Formats:
E <inputfile> <outputfile>
D <inputfile> <outputfile>
T <testfile_with_commands>
or type Q for quiting

Enter command > D ACTGhandout.huf ACTGhandoutRecover.txt

Decoding ACTGhandout.huf ACTGhandoutRecover.txt
FrequencyTable is=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 9, 0, 2, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
Number of bytes in input : 5
Number of bytes in output: 16
```