Computer science problems.

About the problems. The theme of this year's problems is exploring Huffman coding, specifically whether files encoded this way can be editable.

What you need to do. For these problems we ask you to write a program (or programs), as well as write some "paper-and-pencil" solutions (use any text editor that you see fit, or scan an actual handwritten solution; convert the result to pdf format if possible).

You may use any programming language you want for your programs, as long as its full implementation is available at no cost and with an easy installation for both a Mac and Windows (free trial versions do not qualify). It is best to implement each problem as a separate function so that we can run them separately. We will be looking for the following in your submissions:

- Correct code that we can run. You need to send us all your code files, including the header files for languages like C++. If you are using standard libraries, make sure to include all "import" statements, as required by the language you are using. Make sure to send the files under the correct names, including the file extension (.java, .c, etc). Make sure that the file names do not contain any identifying information about you, such as your first or last name.

- Test data for your code that you have used (you can write it in comment or in a separate file). Make sure to test your code well – you don't want it to fail our tests!

- Code documentation and instructions. **Important: do not include your name in comments or in any file names.** If you are submitting your answers to non-code problems in a separate file, also make sure that it does not have your name in the contents or in the file name. The only place where you specify your name is the zip file with your solutions which must be of the form `yourlastname-CS-solution.zip` (replace `yourlastname` by your actual last name). **Make sure that you use zip compression, and not any other one, such as tar**. In the beginning of each file specify, in comments:

  1. Problem number(s) in the file. If you have a file with "helper" functions, mark it as such.
  2. The *programming language*, including the *version* (Java 1.15, for instance), the *development framework* (such as Visual Studio) that you used, unless you were using just a plaintext editor (notepad, emacs, etc), and the *platform* (such as Windows, Mac, Linux)
  3. Instructions for running your program (how to call individual functions, pass the input (if any), etc), either in comments in your program file or as a separate file, clearly named. Please read the instructions for individual problems on the input and output data.
     Input/output files are assumed to be at the default location for your program's project. Make it clear in comments where that is.

4. Some of your code may be commented out if it is not used in the final run of your program. Make sure it is clear what needs to be uncommented to run code for each of the problems.
5. All of your test data.
6. If you were using sources other than the ones listed here (i.e. textbooks, online resources, etc) for ideas for your solutions, please clearly credit these contributions. This is a courtesy to work of others and a part of ethics code for scholars.
7. Make sure that you clearly specify where input files are supposed to be located, provide an example input file and an example of how the file name would be specified in the input. Use relative paths (from the top of the project or from the executable), not absolute paths.

- Clear, understandable, and well-organized code. This includes:

1. Clear separation between problems; comments that help find individual problems and explain how to run the corresponding functions.
2. Breaking down code into functions that are clearly named and described (in comments), using meaningful names for variables and function parameters. Your code should be as self-explanatory as possible. While using comments helps, naming a variable `average` is better that naming it `x` and writing a comment "x represents the average".
3. Minimization of code repetition. Rather than using a copy-paste approach, use functions for repeated code and reuse these functions.
4. Using well-chosen storage structures (use an array or a list instead of ten variables, for instance) and well-chosen programming constructs (use loops or recursion when you can, rather than repeated code).
5. While we are not asking for the fastest program (it's better to make it more readable), you should avoid unnecessary overhead.

Background and problems.

We provide key definitions and ideas. For more details and discussion see [1] or any other textbook that covers Huffman encoding.

In this problem set we will discuss encoding: given an alphabet $L = \{l_1, l_2, \ldots, l_N\}$ of N letters and some other information about L, such as frequency distributions of all letters, provide a unique binary (i.e. using only zeros and ones) encoding for each letter of the alphabet. Let c_i denote the encoding of l_i.

We use $b(c)$ to denote the length of an encoding c (i.e. the number of binary digits in it).

Problem 1. We start by considering an encoding in which each letter of L, i.e. $b(c_i) = b(c_j)$ for all $1 \leq i, j \leq n$. For example, consider the following alphabet of 7 letters and their encodings:

a	b	c	d	e	f	g
000	001	010	011	100	101	110

Note that no character has the encoding 111.

In this case three binary symbols are enough to give each letter its unique encoding. How long would a string of 0s and 1s be in order to encode an alphabet of 26 letters, such as English? What would be the encoding for the letter t? If you are given an alphabet of 33 letters, such as Russian, what would be the length of each letter encoding? Generalize it to an alphabet of N letters for any integer $N > 1$. Please explain all your answers.

Problem 2. Now we start exploring the idea of Huffman coding. Huffman codes are based on the idea that more frequent symbols in an alphabet should be encoded by shorter strings than less frequent letters. Consider the following alphabet with percentages of occurrences of each letter as below:

letter	a	b	c	d	e
frequency, %	44	26	16	8	6

The table shows the letter a on average occurs 44% of the time, the letter b 26% of the time, and so on. Below is an encoding of this alphabet with 0s and 1s using Huffman codes. Note that the more frequent letters have shorter encodings. The letter a, which appears, on average, more than half of the time, is encoded with a single character 0, the letter b with two characters 10. Encodings for `d` and `e` are the same length.

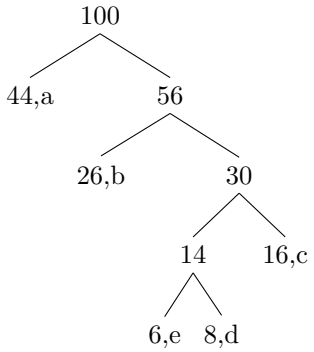
letter	a	b	c	d	e
encoding	0	10	111	1101	1100

Now we can encode words that use this alphabet. For instance, the word *cab* will be encoded as 111010.

The letter codes are of different lengths, but they are constructed so that we know exactly where each letter begins and ends, even though there are no separators between encodings of different letters. This is because no encoding is a prefix (i.e. a beginning) of any other one. For instance, if the encoded word starts with 00, the first letter in it must be b because no other letter starts with 00.

Encode words *ace* and *add* in this alphabet. Decode the following words: 1001101, 1101110011001101.

Problem 3. Huffman encodings of an alphabet can be visually represented as a binary tree. For example, the encoding in problem 2 can be represented like this:



This is a binary tree (each node has at most two children). The left edges correspond to 0 in Huffman encoding, and the right ones to 1. Letters are at the leaves, and sequence of the symbols on the path from the root to a letter is its encoding. For instance, the path to `b` is "right, left", so it's encoded `10`. The path to `d` is "right, right, left, right", so it's encoded `1101`.

Each node (internal or leaf) represents the total percentage of the alphabet that it represents. Each leaf is just the percentage (frequency) of the letter in it. An internal node has the sum of percentages of all letters in its subtree. For instance, 30 in the tree is the sum of percentages of `c, d`, and `e`. The root always corresponds to 100.

Since letters are located at the leaves, no letter is a prefix of another one.

Mirroring any subtree in this tree also produces a valid encoding of the alphabet. By convention we require that:

- Smaller totals of percentages are placed to the left. They may be individual letters (leaves) or internal nodes.
- In case of equal percentages the node with the alphabetically earlier letter anywhere in its subtree is on the left.

Draw a tree for the following encoding and verify that it is a valid Huffman encoding (no letter is a prefix of any other one and no more frequent letter has a longer encoding than a less frequent one):

letter	a	b	c	d	e	f
frequency, %	23	20	19	15	13	10
encoding	01	00	111	110	101	100

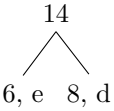
Problem 4. Huffman encodings are constructed by the algorithm described here and in [1]. We also describe it briefly. Note that we follow conventions given in Problem 3.

The algorithm starts by creating a leaf node for each letter. The "weight" of the node is the frequency of the letter. The nodes are placed into a container, such as a set or a priority queue. For instance, in the example described in Problem 3 the 5 leaf nodes are created.

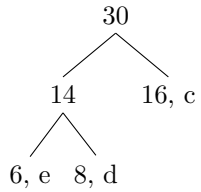
At every step the algorithm takes out two nodes with the smallest combined weight, joins them into one node with the combined weight, and places it back

into the container. The algorithm continues until there is just one node with the weight 100, which is the entire tree.

Continuing with the example, on the first step the two nodes with the smallest weights (`e` with 6 and `d` with 8) will be combined:



On the next step the two smallest nodes are the one we just created and the leaf node for `c` with the weight 16. They are taken out of the container and combined into one node:



After two more steps we obtain the tree given in Problem 3.

Your task for this problem is to implement the algorithm to find Huffman encodings for a given alphabet with frequencies. The program must:

1. Prompt for a file name with the alphabet and frequencies. The file lists each pair of letter/frequency on a separate line. The frequency may be an integer or a may give up to 2 decimal points. No specific order is assumed for the letters, although if two letters have the same frequency, the one that appears earlier is assumed to be alphabetically first. There may be more than 26 letters. All letters are denoted by ASCII symbols (use digits and other symbols for larger alphabets). Example:

   ```
   a 49
   b 25
   c 12
   d 7.52
   e 6.48
   ```

   If the frequencies in the input file don't add up to 100.00 (within some small margin of error to account for rounding errors), the input should be rejected.

2. After prompting for the input file, the program should prompt for the output file name, and write the encodings of the letters in the following format, one letter per line:

   ```
   a 0
   b 10
   c 110
   e 1110
   d 1111
   ```

The output must be sorted by the codes in short-lex order, not in the alphabetical order of the alphabet or by frequencies.

Problem 5. Use the algorithm you developed in Problem 4 to compute Huffman codes for English. Use the frequencies given at

http://practicalcryptography.com/cryptanalysis/letter-frequencies-various-languages/english-letter-frequencies/

You may notice that the frequencies actually add up to 100.01 (you might get an approximate answer due to rounding errors). To remedy this, use 8.54, not 8.55, as a frequency for `a`.

Output the codes into the file `english.txt`. Don't forget to follow the conventions stated in Problem 3.

Problem 6. Implement encoding of a text in a given alphabet. Your program will read:

1. A file name with the text to encode.
2. A file name of the alphabet encodings (it may include symbols in addition to letters, such as digits, but it may not include uppercase letters).
3. A file name of the output file.

The program will work as follows:

1. Convert all capital letters to lower case.
2. Encode the all letters that appear in the alphabet for encoding, ignoring all other symbols (letters that don't appear in the encoding file, spaces and other delimiters, punctuation symbols, etc.)
3. Write out the output to the given file.

Test your program on any English text, include your test data with your submission.

Problem 7. Implement decoding of a text in a given alphabet. Your program will read:

1. A file name with the text to decode.
2. A file name of the alphabet encodings.
3. A file name of the output file.

The program will write out the result of decoding (with no spaces, all letters in lower-case). Test your program on the text from Problem 6. Note that it will not be identical to the starting file because of removal of the non-letter symbols and conversion to lower-case.

Problem 8. In this problem we compare the expected length of encoding with Huffman codes to an encoding of equal length codes, as done in Problem 1.

Your program will read an alphabet with frequencies of letters (just like in Problem 4) and will print out:

1. The length of the encoded text of 10000 letters distributed exactly according to the given frequencies.

2. The *compression factor*, obtained by dividing the above number by the length of the encoding of the same text in which $b(c_i) = b(c_j)$ for all $1 \leq i, j \leq n$.

Problem 9. It is proven that Huffman encoding minimizes the amount of space to store text, assuming that the text has the same frequency distribution as the used in creating the Huffman encoding.

However, the compression factor depends on the frequency distribution. Given an alphabet L of n letters, determine what frequency distribution gives the worst compression factor, compared to equal length codes. Then discuss what frequency distribution would make Huffman encoding the most beneficial. You don't need to give the exact formula - it's ok to just show a few examples and discuss how they can be generalized.

Description of goals and the setup for the next few problems. Now that you are familiar with Huffman encodings, you will explore modifying their storage in memory to allow efficient editing of encoded files. We consider single letter edits and word-level edits, and for each of these levels you will implement three operations: replacement, deletion, and addition.

In order to measure the efficiency of edits you will keep track of the following:

1. The number of bitwise lookups. For instance, if we need to get to a letter at a specific position in a text encoded by Huffman codes we may do so by searching from the beginning and counting letters. In this case we need to look at every bit until we get to the given letter.

2. The number of bit changes. For instance, if we replace a letter a_i by a letter a_j s.t. $b(a_i) = b(a_j)$, we are only replacing $b(a_i)$ binary characters. However, if we replace a_i by a letter a_j s.t. $b(a_i) < b(a_j)$ in a file of contiguous Huffman encodings, in addition to replacing the letter we will need to shift all the subsequent bits forward by $b(a_j) - b(a_i)$. Likewise we would need to shift them backward if $b(a_i) > b(a_j)$.
   Note that bit changes are counted regardless of whether you are replacing a bit by the opposite one or the same one since typically when you are overwriting computer memory you don't check what's currently there.

In addition we measure memory by all the memory taken by the file and all additional space for memory management. We assume that memory addresses are 64 bit integers and that the shortest format for integers (in case you need them) is 32 bits. We also assume that memory can only store binary information, but that there is no need to align files boundaries to any specific boundaries, such as multiples of 8 bits. We also assume that encoded files take up contiguous memory, and that we can add bits at the end without having to move the file.
Important thing to note: We are not asking you to come up with actual bit-level storage, nor will we be measuring the time it takes to perform the edits.

You may simulate the memory using arrays of ints or any other structures. You will be counting operations as you are performing them as if they were on actual bits in memory. You will also compute the amount of memory as if it were actual bits (but see notes above about addresses and other conventions).

Our primary example are Huffman codes for English that you have computed in Problem 5.

Problem 10. Assuming a file of contiguous text encodings, we will consider single letter edits. They may come in the following form: the character number (0-based), the operation `r, d,` or `a` that stand for "replace", "delete", and "add", respectively, followed by a letter for `r` and `a`.

For instance, `23 r h` stands for "replace the character at the position 23 with `h`", `0 d` stands for "delete the first character", and `7 a x` means "add a character `x` at position 7".

Write a function that generates these operations. The length of the text M is the parameter to the function. We assume that the three operations have equal probabilities. Don't implement the actual editing.

Problem 11. Given the text of length M that follows the English frequencies, the English alphabet with its frequencies, and assuming that the positions are uniformly distributed between 0 and $M-1$ and the letter parameters to `r` and `a` are generated by the same English frequencies distribution, compute the average number of bitwise lookups and bit changes (separately) needed to perform each of the three operations on:

- Encodings of equal length.
- Huffman encodings.

Problem 12. Now we extend editing to "words". Since there are no spaces or other delimiters, words will actually be just any consecutive sequences of letters. For replacing or adding a word, these sequences will be randomly generated. Just like in single letter operations, we will have "replace", "delete", and "add". To distinguish them from single letter operations, we will denote them with the capital `R, D,` and `A`, respectively. Note that for `R` and `D` we need to specify the length of the "word" in the text that we are replacing or deleting. Thus the operations will be as follows:

- `R k m c1...cn`, where `k` is the position of the word being replaced, `m` is its length, and `c1...cn` is the "word" to replace it. For instance, `R 71 5 amj` means "replace the 5-letter word at the position 71 with `amj`".
- `D k m`, where `k` is the position of the word being deleted and `m` is its length. For instance, `D 25 3` means "delete 3 letters starting at the position 25".
- `A k c1...cn`, where `k` is the position where the word being added, and `c1...cn` is what's being added. For instance, `A 60 ranju` adds the word "ranju" at the position 60, shifting all letters after it to the right.

You will generate a "word" as follows:

1. Generate the first letter according to the English frequencies distribution.
2. Randomly generate the following: with probability 0.15, the word ends. With probability 0.85, the next letter is added, according to the English frequencies distribution.
3. Repeat the previous step until the end of the word is generated.

To determine the length of the "word" that would be replaced or deleted, perform the same process, just without generating the letters, to determine the length.

For this problem write a program to generate the `R, D, A` commands (with the "words" where needed).

Problem 13. Given the text of length M that follows the English frequencies, and assuming that the positions are uniformly distributed between 0 and $M-m$, where m is the length of deleted or replaced word, and the words and length generated as described in Problem 12, compute or estimate the average number of bitwise lookups and bit changes (separately) needed to perform each of the three operations on:

- Encodings of equal length.
- Huffman encodings.

Problem 14. Your task for this problem is to design and implement a way to make Huffman encoding storage more edit-friendly. The text will still be stored using Huffman codes. You are allowed to add some amount of additional storage (such as an index, extra "growth" memory to expand the encoding to avoid shifting the entire text, or anything else you can think of). However, extra memory should be no more than **5%** of the storage using just Huffman encoding. **Added based on questions:** Note that we count the memory that depends on the amount of the encoded text (say, is proportional to it). Constant memory (such as the address of the data structure itself) doesn't matter.

Your program will read:

- The name of the file with encoded alphabet.
- The name of the file with text. As in Problem 6, you will convert all letters to lowercase, remove all symbols not in the alphabet, and then store this text internally in the setup that simulates your storage. The length of the text will be no more than 50,000 alphabet symbols.
- The name of the file with the edits for the file. The edits will be in the format outlined above. There will be no more than 100 edits. Invalid edits must be rejected (print them out to the standard output).
- The name of the output file.

Your program will **output** the edited Huffman encoding of the file (to be checked for correctness). The output should have no symbols other than 0s and 1s of the Huffman encoding.

In addition your program should **print** the total number of bit lookups, the total number of bit changes, and the total amount of extra memory used (in bits). Any operations performed on any temporary storage are also counted. See the setup for how addresses and integers are counted towards the memory. If you are using any other data types, make some reasonable assumptions about their length or ask.

You may use some of this extra memory to accumulate several edits and perform them all at once. Three conditions should be satisfied:

- The position in an edit is given for the text after all previous edits have been performed. In particular, new edits may overlap with earlier modifications, and should be handled correctly. For instance, if I send a command `R 55 6 htamr` followed by `d 57`, the deleted letter will be `a` in the newly added word.
- You should be able to commit all accumulated edits at any time.
- You may not accumulate more than 5 pending edits at any time.

When submitting your program, make sure to document all your decisions and reasons for them. You might want to discuss whether there is a benefit in using Huffman encodings for editable files, as opposed to fixed-length encodings.

Your program's efficiency will be evaluated based on the following:

- Correctness. If the result (the final output of the encoded file after edits) isn't correct, it doesn't matter how efficient it is.
- The amount of extra memory must not exceed 5%. Again, if it's larger than that then the other measures don't matter.
- The total number of bit changes.
- The total number of bit lookups.
- The total amount of extra memory used.

Code quality, documentation and discussion quality, and similar aspects also play an important role.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms, third edition, 2009.