COSC 2P03 — Assignment 2 — Triglyphic encryption and the Diogenic codex

Diogenes of Sinope was a Greek philosopher with a very colourful life story. Outside of his contributions to Cynicism and existentialism, he was also a logician. One of his more interesting inventions, derived seemingly on a whim, was a mechanism for translating digraphs into triglyphs.

- Simply put, he took a symbols from one alphabet, and devised a scheme for converting them into *combinations* of members of another set of symbols

Through a combinatorial process, he determined a means of creating a *codebook* that could translate between the two. If Plato's accounts are to be believed, Diogenes intended them as a puzzle (similar to a crytogram). If Plato was actually a pompous jerk who constantly misrepresented his contemporaries for self-aggrandizement, it was just a thought exercise. (Supposedly Plato solved it solely through logic, which is a spurious claim at best)

The one interesting thing about the codebook is that it used different numbers of triglyph symbols, depending on how many times the original digraphs showed up. (Since he was using carved stones, he might've just been trying to do 'more with less')

For this assignment, you'll be treating his triglyphic translation as a form of encryption.
Since dealing with digraphs and actual triglyphs in Java would be tedious (and outside the scope of the course), we'll instead use ASCII text, and sequences of binary digits.

- For the sake of simplicity, we won't treat the binary as actual numbers: just patterns of 0s and 1s

Basic Requirements:
For this assignment you need to write two programs:

- A command-line tool that reads in text, creates a Diogenic codex/codebook, and creates an 'encrypted' text file (consisting of the codebook, followed by binary patterns)
- A command-line tool that accepts an encrypted text file, and extracts the codebook to restore the original text (saving it into another file)

And you need to create a document:

- Draw a diagram showing a Diogenic codex (tree), followed by its codebook (matchings), per below

Codebooks:
We'll start with the decryption first, because it's easier.
Suppose we had the following codebook:

```
     1100
!    1101
H    0111
a    000
c    0110
e    010
l    10
o    001
s    111
```

...and the accompanying encrypted message:

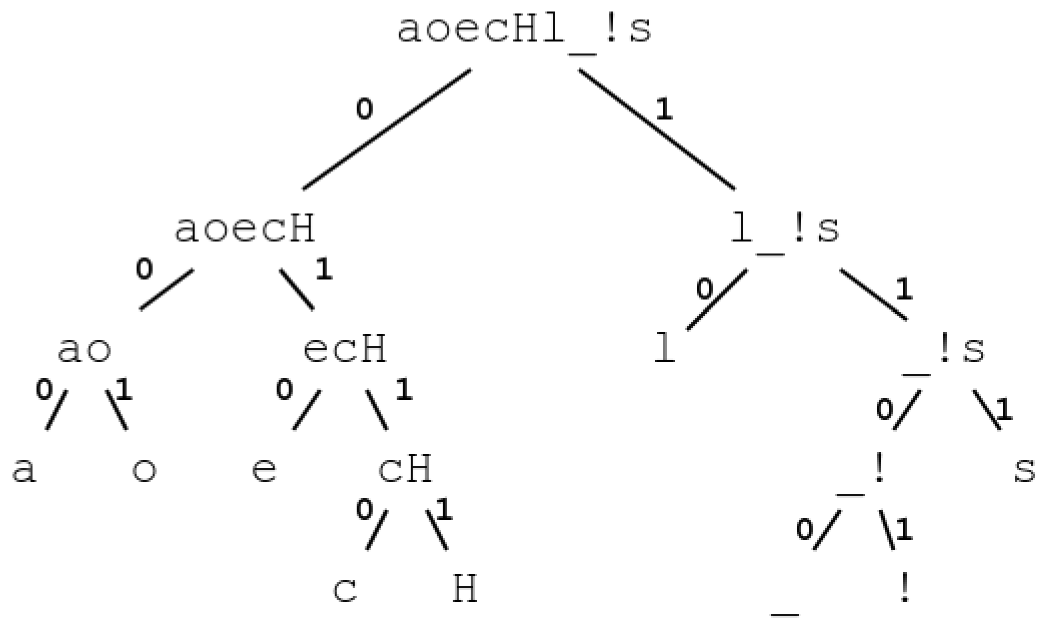
```
0111 010  10   10   001  1100 0110 10   000  111  111  1101
```

Then the final recovered message would be:

```
Hello class!
```

As you can see, the 'codebook' is just the matching between one type of symbol and another. (In this case, a character/string, to a binary pattern)

The Diogenic codex:
The codex itself isn't *terribly* complicated. It's basically just a tree that you traverse to see which sequence of alternate symbols to build up. e.g. (substituting _ for the space):



And so, to find the symbolic pattern for any letter, just follow it down the tree (0s for 'left paths', and 1s for 'right paths'). In this case, a `c` would mean `0110`, and `!` would mean `1101`. The letter `l`, on the other hand, is `10`.

Generating the Diogenic codex:
You might have noticed that some letters have shorter patterns than others. As hinted at earlier, Diogenes probably didn't want to use more stones than needed. So the symbols (letters) that occur the *most frequently* use the *fewest glyphs*.

You start by generating a frequency table:

Letter	_	!	H	a	c	e	l	o	s
Freq.	1	1	1	1	1	1	3	1	2

Since `l` appears the most frequently, it needs the shortest path. Then `s`, and then whatever.

So what do we do? We actually *make* that tree! Specifically, we start with a *forest* of single-node-trees. Each node will have an attached *label* and *frequency*. (For a leaf node, the label's just the letter; for an internal node, it's the labels of all nodes within its subtrees)
You'll be devising something *like* a binary tree for this, though you'll need some extra code.
So you start with the following trees:

```
_:1   !:1   H:1   a:1   c:1   e:1   o:1   s:2   l:3
```

Now, you pick the *two* trees with the *lowest frequencies*, merge them, and re-add them to the pool.
When the numbers are equal, it doesn't matter which you take, so I'll go with the space and exclamation mark:

```
H:1   a:1   c:1   e:1   o:1   s:2   _!:2   l:3
                                    /  \
                                 _:1    !:1
```

Notice that the frequency of a node includes the *sums* of its subtrees. Let's go for `c` and `H`:

```
a:1   e:1   o:1  cH:2  s:2   _!:2  l:3
                 /   \       /   \
              c:1     H:1 _:1     !:1
```

So long as we always pick the two trees with the lowest frequencies, it doesn't matter how else we decide. So next, we could pick `ae`, `ao`, or `eo`. Let's go with `ao`:

```
e:1  ao:2        cH:2  s:2   _!:2  l:3
     /   \       /   \       /   \
  a:1     o:1 c:1     H:1 _:1     !:1
```

Now, we must pick `e`, but we can pick anything other than `l` for the other. Let's go with `cH`:

```
    ao:2  s:2   _!:2    l:3  ecH:3
   /   \       /   \         /   \
a:1     o:1 _:1     !:1   e:1     cH:2
                                 /   \
                              c:1     H:1
```

We're down to only five trees! Making progress! Let's pick `_!` and `s`:

```
    ao:2       ecH:3        l:3    _!s:4
    /   \       /   \               /   \
a:1     o:1 e:1     cH:2        _!:2    s:2
                    /   \       /   \
                c:1     H:1 _:1     !:1
```

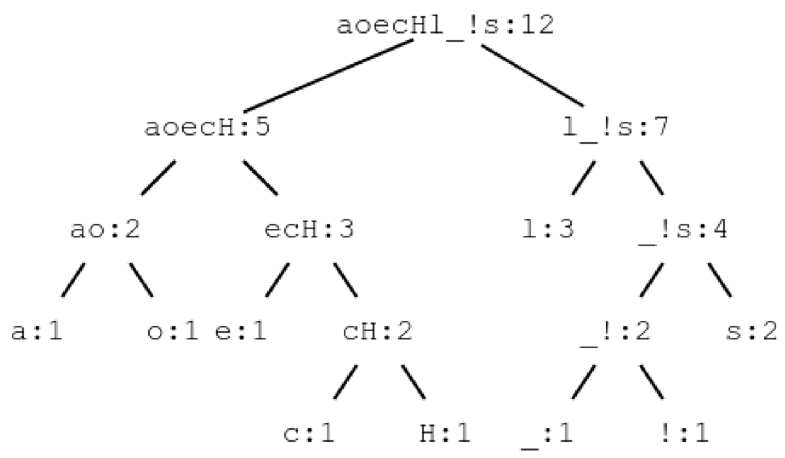
We *need* to pick `ao` next; let's pick `ecH` to go with it:

```
l:3    _!s:4            aoecH:5
        /   \           /     \
    _!:2    s:2     ao:2       ecH:3
    /   \           /   \      /   \
_:1     !:1     a:1     o:1 e:1    cH:2
                                    /   \
                                c:1     H:1
```

We must pick `l` and `_!s`:

```
        aoecH:5                 l_!s:7
         /    \                  /   \
    ao:2       ecH:3          l:3    _!s:4
    /   \       /   \                /   \
a:1     o:1 e:1    cH:2          _!:2    s:2
                    /   \        /   \
                c:1     H:1   _:1     !:1
```

Of course, there are only two left, so we pick those:



And there we go! Since there's only one tree left, we're done!
By this point, you should have a reasonable guesstimate as to why we kept picking the tree with the *lowest* frequency: as we kept merging the trees, those infrequent letters ended up way down in the farthest leaves.

The only major question remaining is: how do we easily *accomplish* this?

- You'll probably make your own variation of a Node class to represent each of these label/frequency tokens. For the sake of comparison, they should probably be *Comparable*...
  - You'll need code *somewhere* that can follow these paths to leaves, to build up/return a binary pattern
- You'll need some sort of data structure that can readily prioritize one tree over another
  - Do we have any data structure that's good for that?

Encryption: `Encrypter.java`

So long as you're okay with each component of above, the rest should be pretty simple.

- Write a program that, based on some text input, creates a *frequency table* of possible characters
  - The input will come in an ASCII file, so you'll definitely never need to worry about more than 256 possible unique characters
  - The input text will *always* be *exactly* one line of text
    - It could be a monstrously-long line of text, including thousands of words, but you can still just use a single `nextLine` to read it
    - If you need individual characters, don't forget that String has a `.charAt()` and `.length()`; and `.toCharArray()`
- Use that frequency table to generate the initial forest of single-character-label trees
  - (Technically, you'll almost certainly still want the labels to be Strings, to simplify the code)
- Perform the algorithm above to create the Diogenic codex (tree)
- Export the codebook to the *output file*
  - The output file will also just be plaintext
  - All you need to do is run through your frequency table, and put out a single line each of: character, a tab, and the binary pattern
    - You can get the binary pattern by walking through the tree (above)
    - Only output entries where the corresponding table entry had a frequency above zero
- The marker between the codebook and the encrypted message is just three dashes: `---`
- Output the corresponding binary patterns of each character in the message, tab-separated

This is meant to be used command-line, so that means you need the option of *command-line arguments*!
Running it with no arguments assumes an input of `testing.txt` and an output of `encrypted.txt`.
With one argument, assume that's the input filename (with an output still of `encrypted.txt`).
With two arguments, they're the input and output filenames.

Decryption: `Decrypter.java`

This one's pretty straightforward, right? It's pretty much explained on Page 1. The only catches:

- The codebook and encrypted message are obviously separated by a line of `---`, per above
- The same command-line parameters apply as above, but with default names of `encrypted.txt`, and `recovered.txt`

Writeup:

To ensure you're comfortable with making *diagrams*, you need to include a reasonably-illustrated writeup. (This document qualifies as adequately 'reasonable', so you don't need to go crazy with it)

Here's what you need to do:

- Submit both a marker-friendly `.pdf`, *and* the files you used to create it (`.docx`, `.tex`, whatever)
- It must contain:
  - A sentence describing your favourite television show (or movie, or imaginary theatre production starring only llamas). Include punctuation, and several words, but remember: only *one* line
  - A frequency table of that sentence
  - A set of diagrams like above of building a possible tree from that frequency table
    - (It's *far* easier than it seems. You're welcome to just use your program to get the codex first, reverse-engineer a tree from that, and then pretend to be building-up said tree)
  - Your name, student number, and username
- Your diagrams must be proper, by which I mean don't just draw it in MS Paint
  - You ***must*** use: https://app.diagrams.net/ (or its equivalent *draw.io* free downloadable program)
    - You *must* **export** from that tool; not use your phone, screenshot, etc.
    - I'm hoping that, by pointing you in the direction of a quality (free) productivity tool, nobody's going to make me regret that…

General:

To clarify a few elements of the above:

- Put the `Encrypter` and `Decrypter` into an `enc` package
- Stick whatever you're using as general storage into a suitably-named package (*not* `enc`)
  - Remember: you're allowed to use code *I* gave you without citation
- Make sure you create your IntelliJ project properly. It should include both packages
  - You're going to need two Run Configurations:
    - One for the encrypter, and one for the decrypter
    - I'll show you how to do this, e.g. **right after our midterm**

Reading from a file is trivial, considering you've already used Scanner. e.g.:

```
Scanner input=null;
try {
	input=new Scanner(new File(infilename));
}
catch (IOException ioe) {System.out.println("Dernit!");} //don't use Dernit
```

After that, it's just like reading from the console.

We haven't done file output, but you can use this if you like:

```
private PrintWriter openFileForSave(String filename) {
	try {
		return new PrintWriter(new BufferedWriter(new FileWriter(filename)));
	}
	catch (IOException ioe) {System.out.println("Ah dangit.");}
	return null;
}
```

You don't *have* to, but it's stupid-easy, because all you'll need are `.print` and `.println`.

- In case you forgot: a *tab* is just `\t`

Submission:
Create `.pdf` output of sample executions of your program. **Zip** those, along with all source, development, and project files used to write the solution, the writeup, etc., and submit through Sakai.

Reminder: This is technically six pages long, so there's a lot to read. If you just skim through and ignore actual requirements, you'll get a zero. The marker won't put more work into grading than you put into your assignment.
Include *all* of the necessary files, make your program run as mandated, and don't write anything absurdly complicated or ridiculously.