Assignment 2: Huffman tree compression

Learning goals

After completing this assignment, you will be able to:

- operate with tree data structures in a real-world application context
- implement recursive operations on trees (both non-mutating and mutating)
- implement the algorithm to generate a Huffman tree, taking design decisions with efficiency in mind.
- implement building blocks for the algorithm to compress data with a Huffman encoding
- implement building blocks for the algorithm to decompress data with a Huffman encoding
- assess possible improvements to a sub-optimal tree and implement the changes needed to generate an improved tree

Introduction

Data compression involves reducing the amount of space taken up by files. Anyone who has listened to an *MP3* file or extracted a file from a *zip* archive has used compression. Reasons for compressing a file include saving disk space and reducing the time required to transfer the file to another computer.

There are two broad classes of compression algorithms: **lossy** compression and **lossless** compression. Lossy compression means that the file gets smaller, but that some information is lost. MP3 is a form of lossy compression: it saves space by throwing away some audio information, and therefore the original audio file cannot be recovered. Lossless compression means that the file gets smaller **and** that the original file can be recovered from the compressed file. FLAC is a form of lossless audio compression: it can't save as much space as MP3, but it does let you perfectly recover the original file.

In this assignment, we'll b sless kind of compression called Huffman coding. When you're finished the assign to compress a file, and then decompress that file to get back the original file.

Before you start, please **r Acknowledgment section of this handout carefully.** This includes your declara e not committed an academic offense and has to be submitted along with your assignment. **Failure to include the `plagiarism.txt` file in your submission will result in a 0 in the assignment.**

Background

Fixed-Length and Variable-Length Codes

Suppose that we had an alphabet of four letters: **a**, **b**, **c**, and **d**. Computers store only 0s and 1s (each 0 and 1 is known as a **bit**), not letters directly. So, if we want to store these letters in a computer, it is necessary to choose a mapping from these letters to bits. That is, it's necessary to agree on a unique code of bits for each letter.

How should this mapping work? One option is to decide on the length of the codes, and then assign letters to unique codes in whatever way we wish. Choosing a code length of 2, we could assign codes as follows: **a** gets **00**, **b** gets **01**, **c** gets **10**, and **d** gets **11**. (Note that we chose a code length of 2 because it is the minimum that supports an alphabet of four letters. Using just one bit gives you only two choices — **0** and **1** — and that isn't sufficient for our four-letter alphabet.) How many bits are required to encode text using this scheme? If we have the text **aaaaa**, for example, then the encoding takes 10 bits (two per letter).

Another option is to drop the requirement that all codes be the same length, and assign the shortest possible code to each letter. This might give us the following: **a** gets **0**, **b** gets **1**, **c** gets **10**, and **d** gets **11**. Notice that the codes for **a** and **b** are shorter than before, and that the codes for **c** and **d** are the same length as before. Using this scheme, we'll very likely use fewer bits to encode text. For example, the text **aaaaa** now takes only 5 bits, not 10!

Unfortunately, there's a catch. Suppose that someone gives us the encoding **0011**. What text does this code represent? It could be **aabb** if you take one character at a time from the code. However, it could also equally be **aad** if you break up the code into the pieces **0** (**a**), **0** (**a**), and **11** (**d**). Our encoding is ambiguous! … Which is really too bad, because it seemed that we had a way to reduce the number of bits required to encode text. Is there some way we can still proceed with the idea of using variable-length codes?

To see how we can proceed, consider the following encoding scheme: **a** gets **1**, **b** gets **00**, **c** gets **010**, and **d** gets **011**. The trick here is that no code is a prefix of any other code. This kind of code is

called a **prefix code**, and it leads to unambiguous decoding. For example, **0000** decodes to the text **bb**; there is no other possibility.

What we have here is seemingly the best of both worlds: variable-length codes (not fixed-length codes), and unambiguous decoding. However, note that some of our codes are now longer than before. For example, the code for **c** is now **010**. That's a three-bit code, even longer than the 2-bit codes we were getting with the fixed-length encoding scheme. This would be OK if **c** letters didn't show up very often. If **c** le n we don't care much that they cause us to use 3 bits. If **c** letters **did** show up ofte ecause we'd be spending 3 bits for every occurrence of **c**.

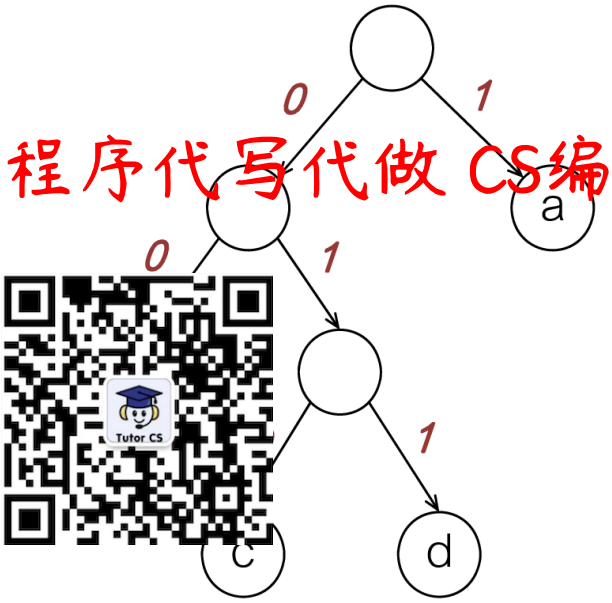
In general, then, we want s for frequently-occurring letters and longer codes for letters that show up less o f the frequency of **a** in our text is 10000 and the frequency of **b** in our text i t the code for **a** would be much shorter than the code for **b**.

Goal of Algorithm**

Our goal is to generate a best prefix code for a given text. The best prefix code depends on the frequencies of the letters in the text. What do we mean by ``best'' prefix code? It's the one that minimizes the average number of bits per symbol, or, alternately, the one that maximizes the amount of compression that we get. If we let **C** be our alphabet, **f(x)** denote the frequency of symbol **x**, and **d(x)** denote the number of bits used to encode **x**, then we're seeking to minimize the sum $\sum_{x \in C} f(x)d(x)$. That is, we want to minimize the sum of bits over all symbols, where each symbol contributes its frequency times its length.

Let's go back to our four-letter alphabet: **a**, **b**, **c**, and **d**. Suppose that the frequencies of the letters in our text are as follows: **a** occurs 500 times, **b** occurs 200 times, **c** occurs 100 times, and **d** occurs 100 times. A best prefix code for these frequencies turns out to be the one we gave earlier: **a** gets **1**, **b** gets **00**, **c** gets **010**, and **d** gets **011**. Calculate the above sum and you'll get a value of 1600. There are other prefix codes that are worse; for example: **a** gets **110**, **b** gets **0**, **c** gets **10**, and **d** gets **111**. Calculate the sum for this and convince yourself that it is worse than 1600!

Huffman's algorithm makes use of binary trees to represent codes. Each leaf is labeled with a symbol from the alphabet. To determine the code for such a symbol, trace a path from the root to the symbol's leaf. Whenever you take a left branch, append a **0** to the code; whenever you take a right branch, append a **1** to the code. The Huffman tree corresponding to the best prefix code above is the following:



Huffman's algorithm generates a tree corresponding to a best prefix code. You'll see proofs of this fact in future courses; for now, you'll implement Huffman's algorithm.

Preliminaries

For this assignment, we're asking you to do some background reading on several topics that are not explicitly taught in the course. (We don't often give students sufficient opportunity to research on their own and build on the available knowledge. This assignment is an attempt to remedy this.) Here's what you'll want to learn:

- Python `bytes` objects. We'll be reading and writing binary files, so we'll be using sequences of bytes (not sequences of characters). `bytes` objects are like strings, except that each element of a `bytes` object holds an integer between 0 and 255 (the minimum and maximum value of a byte, respectively). We will consider each byte in a decompressed file as a symbol.
- Binary numbers. Background on binary numbers will be useful, especially when debugging your code. Although it's not necessarily crucial for this assignment, you might want to also familiarize yourselves with "endianness" (little endian in particular), to get an idea of how computers store a sequence of bytes in memory.

You are strongly encouraged to find and use online resources to learn this background material. Be sure to **cite your sources** in your assignment submission; include Python comments giving the locations of resources that you found useful. Remember that while you are encouraged to explore and understand such topics conceptually, all code you write must be your own. If you copy someone else's code (even bits and pieces of code, whether from online sources or from other students), that constitutes an academic offence and you will jeopardize your academic status.

Huffman's Algorithm

Huffman's algorithm assigns codes to symbols based on their relative frequencies in the file being compressed. The Huffman code for each symbol is variable-length, and prefix-free (a symbol's Huffman code cannot be the prefix of another symbol's Huffman code), as discussed above in the handout. The symbols in the file being compressed are at the leaves of the Huffman tree.

The Huffman algorithm for building the tree that we describe below, tells you what you need to do, conceptually. You must think of efficient ways to implement this algorithm, while following the specs laid out in the handout and the starter code. Please make sure to read the docstrings and type contracts for the functions that you have to implement, and which we will discuss in the next sections.

You MUST NOT discuss with others your ideas on efficient ways to build the Huffman tree, helper methods, or additional data structures that you might consider using. Such design decisions must be entirely your own, and are [illegible] goals of this assignment.

Conceptual algorithm for b[illegible]:

1. Calculate the symbol [illegible]le being compressed. Remember that we operate with any files, so the symb[illegible]arily human-readable text, which means that you will have to read the file b[illegible]s a binary file.
2. Take the lowest two frequency symbols and combine them into a Huffman tree. The lowest frequency symbol out of the two, must be on the left. The root of this Huffman tree has the combined frequency of the two symbols (their sum). This combined frequency, along with the frequencies of any remaining symbols that are not part of a Huffman tree yet, are the input for step 3.
3. Repeat the process from step 2., considering the input that you got from step 2, until all symbols are incorporated into a single Huffman tree. The resulting Huffman tree's root will have the combined frequency of all symbols.

Please revisit the in-class activity worksheet, as well as the lecture slides, if you need additional visual examples of this algorithm.

Special cases that you might be wondering about:

a.: *If we had a table of frequencies with multiple symbols having the same frequency, how would we construct the tree to make it unique and pass the tests, since there are multiple ways to combine the symbols with the same frequency?*

In this particular case of multiple symbols with the same frequency, here are a few strategies you need to use, to get the exact behaviour from the doctests (e.g., from tree_to_bytes, for instance):

- if you have more than two minimums, pick among them in whatever order the dictionary keys are in when you look through them.
- consider symbols that have not been "merged" yet, before other "non-symbol" trees represented in the frequency table
- if you have more than two minimums and they're all "non-symbol" trees, take the earliest formed two trees with that same minimum, so to speak.

b.: What if I have a frequency dictionary with just one symbol-frequency pair?

This can only happen if the input you're compressing has a single symbol. Your remaining code will need Huffman trees with at least two leaves, so in this particular case, just insert a leaf with a dummy symbol (different from the one in your dictionary). You have 255 other choices…

The reason you can use some dummy value is because after all, you shouldn't find anything encoded with this "path" in the tree anyway when you do the decompression. So both the single symbol and the dummy one are still stored in leaf nodes, and the non-dummy leaf can be either to the left or right of the root. The dummy leaf should have a frequency of 0. You should carefully consider the

specifications, *and* carefully read over examples from the doctests (see build_huffman_tree for example).

This should also give you everything you need to know on how to approach building the tree, although **the specifics of the implementation are something that you have to come up with on your own and not discuss with others**.

Your Task

Download the starter code[illegible]umber 7.

Your task is to complete th[illegible]ress.py. Ultimately, you will be able to call the compress_file and decom[illegible]ns to compress and decompress a file, respectively.

The code in compress.py uses the HuffmanTree class from huffman.py, which represents Huffman tree objects. Huffman trees are used to compress and decompress data. The utils.py file contains some utilities for operating with bytes and bits, as well as a ReadNode class, which is the representation of a node's contents in a compressed file. This will be necessary to reconstruct the HuffmanTree from the ReadNodes stored in the file.

We now give a suggested order for implementing the functions in compress.py.

Building the Tree

- Implement build_frequency_dict. This function generates and returns a frequency dictionary.
- Implement build_huffman_tree. This function takes a frequency dictionary and returns a HuffmanTree corresponding to a best prefix code. There's a tricky case to watch for here: you have to think of what to do when the frequency dictionary has only one symbol!
- Implement get_codes. This function takes a Huffman tree and returns a dictionary that maps each byte in the tree to its code. The dictionary returned by get_codes is very useful for compressing data, as it gives the mapping between a symbol and the encoding for that symbol.
- Implement number_nodes. This assigns a number to each internal node. These numbers will be written when compressing a file to help represent relationships between nodes.
- Now is a good time to implement avg_length. This function returns the average number of bits required to compress a symbol of text.

Compressing Data

Now we know how to generate a Huffman tree, and how to generate codes from that tree. Function compress_bytes uses the given frequency dictionary to generate the compressed form of the provided text.

Every byte comprises 8 bits. For example, **00001010** is a byte. Recall that each symbol in the text is mapped to a sequence of bits, such as **00** or **111**. compress_bytes takes each symbol in turn from text, looks-up its code, and puts the code's bits into a byte. Bytes are filled-in from bit 7 (leftmost bit) to bit 0 (rightmost bit). Helper function bits_to_byte will be useful here. You might also find it helpful to use byte_to_bits for debugging; it gives you the bit-representation of a byte.

Note that a code may end up spanning multiple bytes. For example, suppose that you've generated six bits of the current byte, and that your next code is **001**. There's room for only two more bits in the

current byte, so only the **00** can go there. Then, a new byte is required and its leftmost bit will be **1**.

Implement `compress_bytes`.

Writing a Compressed File

`compress_bytes` gives us the compressed version of the text. Suppose that you send someone a file containing that compressed text. How can they decompress that result to get back the original file? Unfortunately, they can't! They don't have the Huffman tree or codes used to compress the text, so they won't know how the bytes in the compressed file correspond to text in the original file.

This means that some representation of the tree must also be stored in the compressed file. There are many ways to do this: we'll use one that is suboptimal but hopefully instructive (DanZ at UTM recalls the technique from old PC games who used Huffman coding to compress a game onto a single disc).

Follow along in function `compress` throughout this discussion.

Here is what we generate so that someone can later recover the original file from the compressed version:

1. The number of internal nodes in the tree
2. A representation of the Huffman tree
3. The size of the decompressed file
4. The compressed version of the file

Functions are already available (in starter code or written by you) for all steps except (2). Here is what we do for step 2, in function `tree_to_bytes`:

- Internal nodes are written out in postorder.
- Each internal node of the tree takes up four bytes. Leaf nodes are not output, but instead are stored along with their parent nodes.
- For a given internal node *n*, the four bytes are as follows. The first byte is a 0 if the left subtree of *n* is a leaf; otherwise, it is 1. The second byte is the symbol of the left child if the left subtree of *n* is a leaf; otherwise, it is the node number of the left subtree. The third and fourth bytes are analogous for the right subtree of *n*. That is, the third byte is a 0 if the right subtree of *n* is a leaf; otherwise, it is 1. The fourth byte is the symbol of the right child if the right subtree of *n* is a leaf; otherwise, it is the node number of the right subtree.

Implement `tree_to_bytes`.

Decompressing Text

There are two functions that work together to decompress text: a `generate_tree_xxx` function to generate a tree from its representation in a file, and a `decompress_bytes` function to generate the text itself.

First, the functions to generate a Huffman tree from its file representation. Note that you are being asked to write two **different** `generate_tree_xxx` functions. Once you get one of them working, you can successfully decompress text. But we're asking for both, so don't forget to do the other one!

- `generate_tree_general` takes a list of `ReadNode`s representing a tree in the form that it is stored in a file. It reconstructs the Huffman tree and returns the root `HuffmanTree` object. This function assumes **nothing** about the order in which the nodes are given in the list. As long as it is given the index of the root node, it will be able to reconstruct the tree. This means that the tree nodes could have been written in postorder (as your assignment does), preorder, inorder, level-order, random-order, whatever. Note that the number of each node in the list corresponds to the index of the node in the list. That is, the node at index 0 in the list has node-number 0, the node at index 1 in the list has node-number 1, etc.
- `generate_tree_postorder` takes a list of `ReadNode`s representing a tree in the form that it is stored in a file. It reconstructs the Huffman tree and returns the root `HuffmanTree` object. This function assumes that the nodes are given in the list in postorder (i.e., in the form generated by your compression code). Unlike `generate_tree_general`, you **are not** allowed to use the node numbers stored in the `ReadNode`s to reconstruct the tree (in fact, these node numbers may not even be set correctly — check the docstring for an example). That is, if byte 1 of an internal node is 1 (meaning that the left subtree is not simply a leaf), then you are not allowed to look at byte 2. Similarly, if byte 3 is 1 (meaning that the right subtree is not simply a leaf), then you are not allowed to look at byte 4.

To test these two functions as you write them, notice that `tree_to_bytes`, combined with the `bytes_to_nodes` utility that we've given you in `huffman_utils.py`, generates tree representations in exactly the form required by `generate_tree_postorder`. To test `generate_tree_general`, things are not so simple. You have no function that generates a tree in anything except postorder. And `generate_tree_general` is supposed to work no matter what, as long as you have the index of the root node in the list. You could, for example, write new `number_nodes` functions that number the trees in different ways (e.g., preorder, random-order, etc.), and write corresponding `tree_to_bytes` functions that write out the tree according to the order specified by the numbering.

Next, the function to decompress text using a Huffman tree:

- `decompress_bytes` takes a Huffman tree, a compressed text, and number of bytes to produce, and returns the decompressed text. There are two possible approaches: one involves generating a dictionary that maps codes to symbols, and the other involves using the compressed text to traverse the tree and discovering a symbol whenever a leaf is reached.

Another Function

For more practice, we are requiring you (as part of your mark) to implement `improve_tree`. This has nothing to do with the rest of the program, so it won't be called by any of your other functions. It is still a required function for you to write, however.

Suppose that you are given a Huffman tree and a frequency dictionary. We say that the Huffman tree is suboptimal for the given frequencies if it is possible to swap a higher-frequency symbol lower in the tree with a lower-frequency symbol higher in the tree.

- `improve_tree` performs all such swaps, improving the tree as much as possible without changing the tree's shape. Note, of course, that Huffman's algorithm will never give you a suboptimal tree, so you can't hope to improve those trees. But you **can** imagine that someone gives you a suboptimal tree, and you want to improve it as much as possible without changing it's shape.

- Note that you should solve this function **only** by swapping symbols via valid swaps (as described above), and not doing anything else like building another huffman tree and copying over symbols.

Testing Your Work

In addition to the doctests provided in `compress.py`, you are encouraged to test your functions on small sample inputs. Make sure that your functions are doing the right thing in these small cases. Finding a bug on a huge e at you have a bug but doesn't give you much information about what is causing the

Once you've suitably teste ou can try the compressor and decompressor on arbitrary files. You'll be ha u can compress and decompress arbitrary files of any type — although the amou that you get will vary widely!

When you run `compress.py`, you have a choice of compressing or decompressing a file. Choose an option and then type a filename to compress or decompress that file. When you compress file `a`, a compressed version will be written as `a.huf`. Similarly, when you decompress file `a.huf`, a decompressed version will be written as `a.huf.orig`. (We can't decompress file `a.huf` back to simply `a`, because then your original `a` file would be overwritten!)

We've provided some sample files to get you started:

- `Homer-Iliad.txt` and `JulesVerne-MysteriousIsland.txt` are two public-domain books from Project Gutenberg (which you should consider reading if you haven't already, even if it's outside the scope of CSC148). Text files like this compress extremely well using Huffman compression.

- `parrot.bmp` is a picture of a parrot in bitmap (bmp) format. This is a decompressed image format and so it compresses very well. We're also including a couple of other bmp images (`mandelbrot_set.bmp` and `julia_set.bmp`) representing fractals from the Mandelbrot set and Julia set.

- `parrot.jpg` is the same picture of a parrot, but in jpg format, which should compress poorly with the Huffman compression, since jpg is already a compressed format for images.

- `bensound-buddy.mp3` and `bensound-sunny.mp3` are two songs (courtesy of bensound.com) in mp3 format, which compresses terribly with Huffman compression.
- `bensound-buddy.wav` and `bensound-sunny.wav` are the same two songs, except in wav format (as you'll notice wav files are much larger, since they are raw audio files, while mp3 files are compressed lossy formats). These files should also compress poorly, with Huffman compression at least (if you're curious how to do better lossless compression for audio files, you can look up the FLAC format).

Throw other files at your compressor and see what kind of compression you can get!

Thorough testing with pytest

The doctests that we've given you do **not** fully characterize what it means for a function to be correct. They're just a starting point for testing. You **must** write rigorous pytest tests to help increase your confidence in your functions. Although any tests that you write are **not** being marked, your submission will be thoroughly tested so we do expect you to come up with lots of **carefully-considered** property tests and unit tests of your own. You may not share your tests with others, as this is part of your responsibility in any assignment. Sharing tests would constitute plagiarism as well.

Sample tests

To help you with testing, we're providing you with some sample tests. In file `test_huffman_properties_basic.py` (see the starter code), we have provided some **property tests** that you can run on your code as you start completing functions. As a reminder, `hypothesis` automatically generates tons of examples that try to disprove the correctness of your function. Please revisit the prep notes on hypothesis testing if necessary, and the hypothesis documentation where needed.

Disclaimer: These sampl arting point and are in no way comprehensive! Remember that one of the e course is thorough testing, so you must write your own pytests, coming up w nsure that your code is rigorously tested.

Polish!

Take some time to polish up. This step will improve your mark, but it also feels so good. Here are some things you can do:

- Pay attention to any violations of the "PEP8" Python style guidelines that PyCharm points out. Fix them!
- In the module you will submit, run the provided PyTA call to check for flaws. Fix them!
- Check your docstrings to make sure they are precise and complete and that they follow the conventions of the function design recipe and the class design recipe.
- **As a reminder any helpers you create must be private! Remember the privacy conventions!**
- Make sure to follow the assignment guidelines found on the assignments main page
- Look for places where you can make improvements to your code to make it more efficient. You shouldn't sacrifice readability, but you shouldn't have your program doing lots of unnecessary work either.
- Read through and polish your internal comments.
- Remove any code you added just for debugging, such as print statements.
- Remove any `pass` statement where you have added the necessary code.
- Remove the "TODO"s wherever you have completed the task.
- Take pride in your gorgeous code!

Plagiarism Acknowledgment

You **must** read the following acknowledgment **carefully** and submit a `plagiarism.txt` file, with the following paragraphs:

"I, _____ [FULL NAME HERE] declare that I have not committed an Academic Offence on this assignment. Specifically:

- I have read the Academic Code of Conduct linked in the course syllabus
- I am aware of what constitutes plagiarism
- No code other than my own (plus starter code) has been submitted
- I have not received any pieces of code from others (including test cases), I have not obtained pieces of code available publicly, nor modified such code to pass as my own work.

- I have not shared any parts of my code with others (including test cases), nor shared specific details on how others may reproduce similar code.
- I did not instruct another classmate on what to write in their assignment code
- I did not receive instruction on specifically what to write in my code, and have come up with the solution myself
- I did not look for assignment solutions online
- I did not attend private tutoring sessions (including in other languages) which are not explicitly sanctioned by UTM w nt was discussed.
- I did not post my own e on places like Github, pastebin, etc. I acknowledge that this declaration is ot include any misrepresentation. I acknowledge that not being able to expl work will result in a zero on this assignment, and that if the code is detected t vere academic penalties will be applied when the case is brought forward to t

Failure to include this file will result in a 0 in the assignment.

Submission Instructions

1. Login to MarkUs and navigate to the assignment 2 submission page.
2. **DOES YOUR CODE RUN?!**
3. Does your code pass PythonTA with no errors?
4. Did you write lots of **different**, **carefully-thought** tests? Do they still pass?
5. Is your code well-documented, following the design recipes?
6. Is your code clean (e.g., avoiding code duplication, using helper functions if appropriate, using ADTs to abstract implementation details if necessary, no chunks of commented code that no longer works or was only there for debugging purposes, etc.).
7. Submit the `compress.py` file and your `plagiarism.txt` file.
8. On any DH lab machine, make a new folder and download your submitted file into it. Test your code thoroughly. *Your code will be tested on the DH lab machines, so it must run in that environment.*. This step will also confirm that you **submitted the right version** of the required files, and didn't **introduce an error at the last moment**, for example by adding a comment of changing a variable name. **IMPORTANT: There is no late submission opportunity for Assignment 2, if you don't have any grace tokens left, so make sure your code runs!**
9. Congratulations, you are finished with your last assignment in CSC148! You are now one step closer to being a wizard/witch who masters parser-tongue. :)

Mathematical & Computational Sciences
UNIVERSITY OF TORONTO
MISSISSAUGA

For general course-related questions, please use the discussion board.
For individual questions, accommodations, etc., please contact
the **course mailing list email as per the piazza instructions**.
Make sure to include CSC148-2023 in the subject, use your official university email to send the message from, and to
state your name and UtorID in the email body.