# A453 Controlled Assessment (Material 1 - string compression)

I will be writing code for this assessment in Python 3.6, using the Python standard library. Code will be written to work as a command line tool, which I will elaborate more on in the controlled assessment. Note that it was actually developed in 3.5, but a Python 3.6-specific feature was added later.

I will include code samples as syntactically highlighted text. I will use the IDLE highlighting scheme for python code, and use a black background for any sample command line commands. Sample program output will be in an unhighlighted monospace font

# Task 1 - Finding the locations of a word in a sentence

## Approach #1: sentence_indices.py (searching through the list for every query)

The task is to “identify the positions where a word occurs in a sentence”. This problem can be broken down into a few smaller algorithms.

The question is theoretically asking to find all indices in a list where an item occurs. This is the first algorithm. The second algorithm is to actually transform a given sentence, in string form, into a usable list.

### Algorithm #1: locations (finding indices of an item in a list)

This algorithm will need to find the locations of a word in a list of words, case insensitively. For case insensitive performance, I will assume that it is given a list of already uppercase words, which will be provided by another function. I am using uppercase as that was the case of the example. This function should return a list of the locations where a word appears. I will define its behaviour in the case of a word not appearing as returning an empty list, rather than some sort of exception, as functionally, this makes more sense in terms of its behaviour - the length of the returned list will be the same as the number of occurrences of the word, for example. Some example cases of use are:

```
>>> locations(["ONE", "TWO", "THREE"], "tWo")
[2]
>>> locations(["ONE", "TWO", "ONE"], "one")
[1, 3]
>>> locations(["ASK", "NOT", "WHAT", "ASK"], "ask")
[1, 4]
>>> locations(["EGGS", "HAM", "SPAM", "SPAM"], "spam")
[3, 4]
>>> locations(["EGGS", "HAM", "SPAM", "SPAM"], "BACON")
[]
```

And this is its desired behaviour in some edge cases:

```
>>> locations([], "one")
[]
>>> locations(["ONE", "TWO", "THREE"], "FOUR")
[]
```

I can achieve this with the following algorithm (in pseudocode):

```
initialise an empty list to contain the indices where the word is
for each word in the list to search:
    if this word is the same as the uppercased target word:
        add the index of the word to the list
```

In Python, this can be done with the following function:

```
def locations(sentence, word):
    matching_indices = []
    for ind in range(len(sentence)):
        if sentence[ind] == word.upper():
            matching_indices.append(ind)
    return matching_indices
```

However, this is simplistic. My code makes use of Pythonic features to improve on this. The first thing my code uses is Python's enumerate function, which returns an iterable of tuples which have the corresponding index and item from the list. This is useful for a situation like this, where I need both the item and its index in a for loop. Together with enumerate, my code uses tuple unpacking, to get the values in the tuple (the index and the item) into the variable ind and i. Using enumerate, the code looks like this:

```
def locations(sentence, word):
    matching_indices = []
    for ind, i in enumerate(sentence):
        if i == word.upper():
            matching_indices.append(ind)
    return matching_indices
```

The last thing it does is pack this logic into a list comprehension. This is a construct in Python that can be used to perform an operation on each item in a list, without the

boilerplate of initialising an empty list and appending each item, which is used in this code. For example, to get the square numbers up to $9^2$, you can do

```
>>> squares = [i**2 for i in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Now, my code doesn't actually need to perform any operation on the items (the indices) but it uses a condition to filter some of them out. This is called a predicate, and can also be used in a list comprehension, to finally bring us back to

```
def locations(sentence, word):
    return [ind for ind, i in enumerate(sentence)
            if i == word.upper()]
```

(Note - This line was going to be exceedingly long, which is undesirable for a number of reasons - it goes against Python's style guide, PEP 8, and it would mean the line would wrap in Google docs, too. Therefore, I've broken up the line myself, to preserve coherency and clarity. Breaking a line in a place like this is permissible in Python because of the implicit line continuation between brackets and parentheses. I will need to do this more often throughout this assessment.)

However, the question also uses 1-based indexing, as I can see from the example that "COUNTRY" appears at indexes 5 and 17, rather than 4 and 16. Python uses 0-based indexing, so I will need to account for this. I could use the option for a uniform operation here, and use the expression (ind + 1), but another approach is to use the optional start parameter for enumerate, giving the starting index. This comes together to our function:

```
def locations(sentence, word):
    return [ind for ind, i in enumerate(sentence, 1)
            if i == word.upper()]
```

This is pretty standard Python. There are no type declarations. However, as this is my controlled assessment, I will be wanting to fully document the type of each variable. Therefore, I am going to use Python 3.6's new fully fledged variable annotation feature - this allows each variable to be given an annotation, and something similar to be done with function signatures. Note that this makes my code entirely incompatible with earlier Python, unless you remove any styles of variable annotation that have not yet been introduced. Along with the typing module, I can define all of my types within the code, like this. No type checking occurs, but it improves the clarity of my code.

```
def locations(sentence: List[str], word: str) -> List[int]:
    ind: int
    i: str
    return [ind for ind, i in enumerate(sentence, 1)
            if i == word.upper()]
```

Note that, as mentioned, no type checking occurs anyway in Python. Aside from that, I am not doing any more checking of the validity of parameters passed to the function. This is

because this would be tiresome to do, greatly reducing the conciseness of my code overall, and it would also cost in efficiency, which, especially later, will be a very important consideration. If I check a number of conditions every time a small function like this is called, my program will end up running very slow. Hence, I will just assume that the function is given correct arguments. If they are not given, the function's behaviour is not defined or guaranteed. This is the idea of "garbage in, garbage out" - my function works so long as the rest of the code also works with it. This does not mean that my code is likely to break, however. If I have written my code properly, the user will not be able to break it at all, because on a system-wide level, the user is unable to compromise any of my units of code. This function, for example, expects a list of uppercase words, which are derived from a user's input, but the user's input is a continuous string. However, the user has no direct access to this function, only my code has, and my code supplies this function with the output of another function which is guaranteed to produce appropriate data for this function. In Python, this is sometimes referred to as the "we are all consenting adults here" philosophy - my function can trust that it will be used properly. I will also demonstrate that on a system wide level, my code works.

## Algorithm #2: clean_sentence (transforming a string into a list of words)

The input is "a sentence that contains several words without punctuation". I will take this to mean a string, composed of letters and whitespace (I make the assumption that the input takes this form). This function will need to split this string into a list of words. Another thing it could do is bring the words into one case - I will do upper case, as the example sentence is also upper case - as the system should not be case sensitive. Here are some example cases of normal behaviour:

```
>>> clean_sentence("OnE TWO three")
['ONE', 'TWO', 'THREE']
>>> clean_sentence("Tom and JErry")
['TOM', 'AND', 'JERRY']
>>> clean_sentence("python is great")
['PYTHON', 'IS', 'GREAT']
```

Here are some edge cases:

```
>>> clean_sentence("One")
['ONE']
>>> clean_sentence("")
[]
>>> clean_sentence(" foo ")
['FOO']
```

In pseudocode, this algorithm could be written as:

```
take the sentence string
transform it to all uppercase
split it by whitespace
```

In Python, this function can be written, quite concisely, like this:

```
def clean_sentence(sentence: str) -> List[str]:
    return sentence.upper().split()
```

It does exactly what the pseudocode says - it first calls the upper() method, which will replace all lowercase characters in the sentence with the corresponding uppercase character. It then calls the split() method, which will, as it says on the tin, split the sentence into a list of string in the sentence. When called without a parameter it splits by whitespace, which is my desired effect.

## Main

In Python convention, the code that actually starts to do things is encapsulated in a main function, which is only called if the Python file is run as a script. This allows for elements of the code to be reused, by importing it as a module. I will be following this convention.

The way I am taking input will be from the command line. This is because I will be designing all of my code in this assessment to be run in the command line. I'm doing this because I work comfortable and productively in the command line, and I can easily manipulate inputs and outputs from units of code from the command line, such as supplying large chunks of text using cat, and doing tests on the output by piping it to other commands.. I can also easily repeat a run of the program (by pressing the up arrow to run the previous command), as opposed to if I were using input to make the user manually enter the data - for testing I would have to keep typing out whatever inputs I want to give my program. It should be noted that the shell being used is not taking any of the computation away from the script. The most I will use the shell for is simple manipulation of data streams - measuring the size of a script's output, supplying the output from one script as the input of another. I might also use the command line to do simple manipulation of files, like using the head command to only supply a script with the first 500 lines.

I will take input of the sentence from sys.stdin. As a quick way to take the words to find the indices of, I will use the command line arguments.

```
def main() -> None:
    sentence: List[str] = clean_sentence(sys.stdin.read())
    print(f"the sentence has been parsed as {sentence}")
    #python 3.5:
    #print("the sentence has been parsed as {}".format(sentence))
    words: List[str] = sys.argv[1:]
    word: str
    for word in words:
        print(f"the word '{word}' appears at the indices "
              f"{locations(sentence, word)}")
        #print("the word '{}' appears at the indices {}"
        #      .format(word, locations(sentence, word)))
```

This also uses a new 3.6 feature - the f-string, also called literal string interpolation. This means that values from my program can be written directly in a special 'f-string' literal and evalutuated when necessary. For some semblance of compatibility, I have in comments the

same functionality implemented with str.format(). As the f-string was too long to fit on one line, I used another Python feature - implicit string concatenation. String literals written consecutively in Python will be concatenated.

I also need to add the following:

```
if __name__ == "__main__":
    main()
```

As said previously, this together forms the Python approach to allowing code to be both modular and run as a script. I can now reuse functions from this script using an import statement, but I can also directly run this script, using python sentence_indices.py, which means that the __name__ variable will be set to “__main__”, causing the main() function to be executed.

I can now call my script from the command line as follows, using the echo command to supply STDIN which will be piped to the program.

```
$ echo "Ask noT wHAT your country can do for you ask what you can do
for your country" |  python sentence_indices.py ask CountRY CAN
```

The resulting output is:

```
the sentence has been parsed as ['ASK', 'NOT', 'WHAT', 'YOUR',
'COUNTRY', 'CAN', 'DO', 'FOR', 'YOU', 'ASK', 'WHAT', 'YOU', 'CAN',
'DO', 'FOR', 'YOUR', 'COUNTRY']
the word ask appears at the indices [1, 10]
the word CountRY appears at the indices [5, 17]
the word CAN appears at the indices [6, 13]
```

The code appears to be working perfectly. I can see that it behaves case insensitively, and that it correctly locates the indices.

Below, I have included the entire script, with documentation. For documentation, I have used the Python convention of the docstring - including a multiline string right after the declaration of each function. This is parsed by python, and put in the __doc__ attribute of the function. It can later also be viewed when using the help() function, for example. I have not yet included any comments as I don’t think my code does anything unintuitive enough to need explanatory comments, and I have explained my code in detail in this document. My code also often has the property of being self documenting - it is redundant to say that clean_sentence(sys.stdin.read()) will clean a sentence read from stdin, for example.

**#sentence_indices.py**

```
"""
A script to find the locations at which a word appears in a sentence, using
1-based indexing, case insensitively.
"""

import sys

from typing import *
```

```
def locations(sentence: List[str], word: str) -> List[int]:
    """
    Find the locations of a string in a list of uppercase strings, case
    insensitively.

    Example usage:
    >>> locations(["ONE", "TWO", "THREE"], "tWo")
    [2]
    >>> locations(["ONE", "TWO", "ONE"], "one")
    [1, 3]
    >>> locations(["ASK", "NOT", "WHAT", "ASK"], "ask")
    [1, 4]
    >>> locations(["EGGS", "HAM", "SPAM", "SPAM"], "spam")
    [3, 4]
    >>> locations(["EGGS", "HAM", "SPAM", "SPAM"], "BACON")
    []

    Parameters:
    sentence - List[str] - list of uppercase strings, as returned by
     clean_sentence, to find the locations in
    word - str - string to find the location of

    Return:
    List[int] - a list of integers representing the locations at which the word
     appears
    """

    ind: int
    i: str
    return [ind for ind, i in enumerate(sentence, 1)
            if i == word.upper()]

def clean_sentence(sentence: str) -> List[str]:

    """
    Clean a string containing a mixed case sentence into a list of uppercase
    words.

    Example usage:
    >>> clean_sentence("OnE TWO three")
    ['ONE', 'TWO', 'THREE']
    >>> clean_sentence("Tom and JErry")
    ['TOM', 'AND', 'JERRY']
    >>> clean_sentence("python is great")
    ['PYTHON', 'IS', 'GREAT']

    Parameters:
    sentence - str - a string containing mixed case words, separated by
     spaces

    Return:
    List[str] - a list of uppercase strings
    """

    return sentence.upper().split()

def main() -> None:
    sentence: List[str] = clean_sentence(sys.stdin.read())
    print(f"the sentence has been parsed as {sentence}")
    #python 3.5:
    #print("the sentence has been parsed as {}".format(sentence))
    words: List[str] = sys.argv[1:]
    word: str
    for word in words:
        print(f"the word '{word}' appears at the indices "
```

```
            f"{locations(sentence, word)}")
        #print("the word '{}' appears at the indices {}"
        #      .format(word, locations(sentence, word)))

if __name__ == "__main__":
    main()
```

The documentation may look like a bit much, but it is good practice, especially for when my code will start to become more complex. Also, having example usage in documentation can be used for testing, as I am just about to talk about.

## Testing

I have already supplies some examples of my code working in what is called a system test. This can show me if my code appears to be working, and can catch most bugs. However, there is a better way to test code - because of the modular way in which I have written my code, I can run tests against each function (unit of code). This is called unit testing, and Python supports a framework for this. Writing more code to test my code is a good idea because it means I can easily re-run tests to boost confidence in my code, using this command:

```
$ python -m unittest discover -v
```

This is using Python's -m switch to run a module in the path. This runs the module directly, so any code in the if __name__ == "__main__": part is run. This handily demonstrates why I do this. It runs unittest, and tells it to "discover" any tests in the directory. It can identify tests because of the way they are written - subclassing unittest.TestCase. I could also run it directly on a module to find any tests there, but is somewhat more efficient to use this command to run all tests at once. Finally, I am using the -v switch to make it verbose, just so I can see the tests it is doing.

Unit testing also enable me to very quickly see which part of the code has gone wrong, where a system test does not.

I have already written about some example test cases for each function in its own "Algorithm" section, so the test section will just show the implementation of the test. I will use one or two examples of normal use, and all the edge cases I can think of, for testing.

The way a unit test is written in Python is pretty simple - you subclass unittest.TestCase, and define test methods for each thing you want to test. unittest.TestCase provides assertion methods to test behaviour - the main one is unittest.TestCase.assertEqual, which can test if two things that are expected to be equal are equal. Using this, I can write my tests.

When I run my tests, I get the following output:

```
test_clean_sentence (test_sentence_indices.TestSentenceIndices) ... ok
test_locations (test_sentence_indices.TestSentenceIndices) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

This tells me that my code is working. In future I will continue to write unit tests in this same way.

**#test_sentence_indices.py**

```
import unittest

from sentence_indices import *

class TestSentenceIndices(unittest.TestCase):
    def test_clean_sentence(self) -> None:
       self.assertEqual(clean_sentence("OnE TWO three"),
                        ["ONE", "TWO", "THREE"])
        self.assertEqual(clean_sentence("One"), ["ONE"])
        self.assertEqual(clean_sentence(""), [])
        self.assertEqual(clean_sentence(" foo "), ["FOO"])

    def test_locations(self) -> None:
        self.assertEqual(locations(["ONE", "TWO", "THREE"], "tWo"), [2])
        self.assertEqual(locations(["ONE", "TWO", "ONE"], "one"), [1, 3])
        self.assertEqual(locations(["EGGS", "HAM", "SPAM", "SPAM"], "BACON"),
                         [])
        self.assertEqual(locations([], "one"), [])
        self.assertEqual(locations(["ONE", "TWO", "THREE"], "FOUR"), [])
```

## Doctesting

Note that there is another framework in Python that can be used for testing - this is doctest. doctest searches through all docstrings, about which I talked earlier, looking for examples of use in the interpreter (with >>>), and verifying that they work. It can be used from the command line like this:

```
$ python -m doctest -v sentence_indices.py
```

With the following output:

```
Trying:
    clean_sentence("OnE TWO three")
Expecting:
    ['ONE', 'TWO', 'THREE']
ok
Trying:
    clean_sentence("Tom and JErry")
Expecting:
    ['TOM', 'AND', 'JERRY']
ok
Trying:
```

```
    clean_sentence("python is great")
Expecting:
    ['PYTHON', 'IS', 'GREAT']
ok
Trying:
    locations(["ONE", "TWO", "THREE"], "tWo")
Expecting:
    [2]
ok
Trying:
    locations(["ONE", "TWO", "ONE"], "one")
Expecting:
    [1, 3]
ok
Trying:
    locations(["ASK", "NOT", "WHAT", "ASK"], "ask")
Expecting:
    [1, 4]
ok
Trying:
    locations(["EGGS", "HAM", "SPAM", "SPAM"], "spam")
Expecting:
    [3, 4]
ok
Trying:
    locations(["EGGS", "HAM", "SPAM", "SPAM"], "BACON")
Expecting:
    []
ok
2 items had no tests:
    sentence_indices
    sentence_indices.main
2 items passed all tests:
   3 tests in sentence_indices.clean_sentence
   5 tests in sentence_indices.locations
8 tests in 4 items.
8 passed and 0 failed.
Test passed.
```

From this I can see that all the examples work. This helps me in two ways - it provides extra testing, and it reassures me that my documentation is correct. However, doctest has a few problems - one of which is very problematic. I will cover this in the next approach to this task.

## Approach #2: hashing_approach.py (precomputing with a dictionary)

There is one more consideration: the question states that the user may continue to input words. An optimisation here to be made is to pre-compute all of the indices, and store these in a dictionary mapping, from words to the list of their indices. Dictionaries have a constant lookup time, so if many many words need to be found, this approach will be faster, although the precomputation step may take some time. In practice for the wording of this question, it will probably not have much of a difference. Also, a user is not likely to query the same word multiple times, but this is where this approach is superior. However, looking at later tasks, optimizing on multiple queries of one word is a relevant technique.

## Algorithm #1: build_dictionary (builds a dictionary)

This algorithm will need to, for every word in the sentence, calculate its locations and store this in a dictionary. Here are some sample cases:

```
>>> build_dictionary(["ONE", "TWO", "THREE"])
{'THREE': [3], 'TWO': [2], 'ONE': [1]}
>>> build_dictionary(["ONE", "TWO", "ONE"])
{'TWO': [2], 'ONE': [1, 3]}
>>> build_dictionary(["CAT", "DOG", "DOG", "CAT"])
{'CAT': [1, 4], 'DOG': [2, 3]}
```

And some edge cases:

```
>>> build_dictionary(["ONE"])
{'ONE': [1]}
>>> build_dictionary([])
{}
```

This algorithm in pseudocode will look like the following:

```
initialise a dictionary to map from words to lists of indices
for each item in the list of words:
    call the previous algorithm to get the list of indices
    push the word and this list to the dictionary
```

In Python, this can be implemented like so:

```
def build_dictionary(sentence: List[str]) -> Dict[str, List[int]]:
    word: str
    return {word: locations(sentence, word) for word in sentence}
```

This again uses a comprehension - but this time a dictionary comprehension, to quickly build a dictionary from key: value pairs.

## Main

I can then add some logic to the main function to build a dictionary and lookup the indices in that, like so:

```
def main() -> None:
   sentence: List[str] = clean_sentence(sys.stdin.read())
    print(f"the sentence has been parsed as {sentence}")
    #print("the sentence has been parsed as {}".format(sentence))
    words: List[str] = sys.argv[1:]
    location_dict: Dict[str, List[int]] = build_dictionary(sentence)
    word: str
    for word in words:
        print(f"the word {word} appears at the indices "
              f"{location_dict[word.upper()]}")
```

```
        #print("the word {} appears at the indices {}"
        #      .format(word, location_dict[word.upper()]))
```

And now, the full script can be called in the same way as the first approach:

```
$ echo "Ask noT wHAT your country can do for you ask what you can do
for your country" |  python sentence_indices.py ask CountRY CAN
```

Again, with the (correct) output:

```
the sentence has been parsed as ['ASK', 'NOT', 'WHAT', 'YOUR',
'COUNTRY', 'CAN', 'DO', 'FOR', 'YOU', 'ASK', 'WHAT', 'YOU', 'CAN',
'DO', 'FOR', 'YOUR', 'COUNTRY']
the word ask appears at the indices [1, 10]
the word CountRY appears at the indices [5, 17]
the word CAN appears at the indices [6, 13]
```

**#hashing_approach.py**

```
"""
A script to find the locations at which a word appears in a sentence, but using
precomputation and hashing.
"""

import sys

from sentence_indices import locations, clean_sentence

from typing import *

def build_dictionary(sentence: List[str]) -> Dict[str, List[int]]:
   """
    Builds a dictionary of precomputed locations for the words in the sentence.

    Example usage:
    >>> build_dictionary(["ONE", "TWO", "THREE"])
    {'THREE': [3], 'TWO': [2], 'ONE': [1]}
    >>> build_dictionary(["ONE", "TWO", "ONE"])
    {'TWO': [2], 'ONE': [1, 3]}
    >>> build_dictionary(["CAT", "DOG", "DOG", "CAT"])
    {'CAT': [1, 4], 'DOG': [2, 3]}

    Parameters:
    sentence - list of strings

    Returns:
    dictionary mapping from strings to lists of integers
    """

    word: str
    return {word: locations(sentence, word) for word in sentence}

def main() -> None:
   sentence: List[str] = clean_sentence(sys.stdin.read())
    print(f"the sentence has been parsed as {sentence}")
    #print("the sentence has been parsed as {}".format(sentence))
    words: List[str] = sys.argv[1:]
    location_dict: Dict[str, List[int]] = build_dictionary(sentence)
    word: str
    for word in words:
        print(f"the word {word} appears at the indices "
```

```
            f"{location_dict[word.upper()]}")
        #print("the word {} appears at the indices {}"
        #      .format(word, location_dict[word.upper()]))

if __name__ == "__main__":
    main()
```

## Testing

**#test_hashing_approach.py**

```
import unittest

from hashing_approach import *

class TestHashingApproach(unittest.TestCase):
    def test_build_dictionary(self) -> None:
       self.assertEqual(build_dictionary(["ONE", "TWO", "THREE"]),
                        {'THREE': [3], 'TWO': [2], 'ONE': [1]})
        self.assertEqual(build_dictionary(["ONE", "TWO", "ONE"]),
                        {'TWO': [2], 'ONE': [1, 3]})
        self.assertEqual(build_dictionary(["ONE"]), {'ONE': [1]})
        self.assertEqual(build_dictionary([]), {})
```

Now that I have written a few tests, this is how a unit test can be run:

```
$ python -m unittest discover -v
```

This automatically discovers test cases. When it is run in the directory with test_sentence_indices.py and test_hashing_approach.py, it has an output like this:

```
test_build_dictionary (test_hashing_approach.TestHashingApproach) ...
ok
test_clean_sentence (test_sentence_indices.TestSentenceIndices) ... ok
test_locations (test_sentence_indices.TestSentenceIndices) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

From this, I can see that all my code for task 1 is working perfectly, and I can and will quickly run this command, periodically, to confirm that my code still works. In future, I will no longer verbosely run unittest, and I will run it individually on each approach.

## Doctesting

Interestingly, doctest no longer works properly at this point, because I have a function that generates a dictionary. The string representation of equivalent dictionary objects is not guaranteed to be the same, which is because there is a nondeterministic factor in the building of a dictionary in Python: dictionaries work by hashing the keys, and Python's

hashing has a random salt, which means the hash is different every time you run the program (this is to stop adversaries from exploiting hash collisions). The ordering in a dictionary is based on this, so it will change every time. This causes doctest to fail as doctest works by string representation rather than value. If I try to run a doctest, I get a result like this:

```
$ python -m doctest -v hashing_approach.py
```

Output:

```
Trying:
    build_dictionary(["ONE", "TWO", "THREE"])
Expecting:
    {'ONE': [1], 'TWO': [2], 'THREE': [3]}
**********************************************************************
File ".\hashing_approach.py", line 13, in hashing_approach.build_dictionary
Failed example:
    build_dictionary(["ONE", "TWO", "THREE"])
Expected:
    {'ONE': [1], 'TWO': [2], 'THREE': [3]}
Got:
    {'THREE': [3], 'ONE': [1], 'TWO': [2]}
Trying:
    build_dictionary(["ONE", "TWO", "ONE"])
Expecting:
    {'TWO': [2], 'ONE': [1, 3]}
**********************************************************************
File ".\hashing_approach.py", line 15, in hashing_approach.build_dictionary
Failed example:
    build_dictionary(["ONE", "TWO", "ONE"])
Expected:
    {'TWO': [2], 'ONE': [1, 3]}
Got:
    {'ONE': [1, 3], 'TWO': [2]}
Trying:
    build_dictionary(["CAT", "DOG", "DOG", "CAT"])
Expecting:
    {'DOG': [2, 3], 'CAT': [1, 4]}
**********************************************************************
File ".\hashing_approach.py", line 17, in hashing_approach.build_dictionary
Failed example:
    build_dictionary(["CAT", "DOG", "DOG", "CAT"])
Expected:
    {'DOG': [2, 3], 'CAT': [1, 4]}
Got:
    {'CAT': [1, 4], 'DOG': [2, 3]}
2 items had no tests:
    hashing_approach
    hashing_approach.main
**********************************************************************
1 items had failures:
   3 of   3 in hashing_approach.build_dictionary
3 tests in 3 items.
0 passed and 3 failed.
***Test Failed*** 3 failures.
```

An example of what it perceives as a failure here is

```
Expected:
    {'ONE': [1], 'TWO': [2], 'THREE': [3]}
Got:
    {'THREE': [3], 'ONE': [1], 'TWO': [2]}
```

This is one nice example of why unit testing is a lot more reliable than doctesting. As I will be working a lot with dictionaries, I will no longer doctest (although I will of course still write documentation).

## Comparing the two approaches:

I can write another short script to test the speed of each approach. I will not be writing tests or as much formal documentation for this script, this is more of a quick way to compare.

This script will simulate a use case. The user supplies the number of tests to do through the command line, and the script generates a list of words chosen randomly from the given sentence. It then tests both approaches, timing both, and storing their results. It prints whether the approaches got the same results, and the respective time for each approach.

Testing on the given “ask not what..” sentence, for a large number of tests, we can see that the hashing approach is a lot faster. Here, for a small number of tests, not much will be apparent as both times will be very fast.

```
$ echo "Ask noT wHAT your country can do for you but what you can do
for your country" | python test_approaches.py 1000000
```

Output:

```
doing 1000000 tests
read words
generated test words
starting naive test
finished naive test
beginning hash test
they agree
naive time was 3.9133
hash time was  0.0776 (0.0001 of which spent building dictionary)
```

The other cases to test are where there is a larger input. Here, the naive approach will be faster for a small number of tests, as the dictionary takes so long to build, but as the number of tests increases the dictionary regains its advantage.

```
$ cat ../text/shakespeare.txt | head -500 | python test_approaches.py
10
```

Note that in Powershell, the head command must be simulated with

```
select -first 500
```

Output:

```
doing 10 tests
read words
generated test words
starting naive test
finished naive test
beginning hash test
they agree
naive time was 0.0075
hash time was  1.9010 (1.9010 of which spent building dictionary)
```

Here, as a sample large input I am using the first 500 lines of the works of Shakespeare (from project Gutenberg). We can see that for a small number of tests, the naive approach is much faster. However, when testing more often:

```
$ cat ../text/shakespeare.txt | head -500 | python test_approaches.py 10000
```

Output:

```
doing 10000 tests
read words
generated test words
starting naive test
finished naive test
beginning hash test
they agree
naive time was 5.6197
hash time was  1.9059 (1.9050 of which spent building dictionary)
```

We can see that the hashing approach is now comparatively very very efficient, as its time has barely gone up while the naive approach has.

This code is not particularly well written, and is not documented or annotated. It is included more as an interest on the side than as a part of the holistic assessment.

**#test_approaches.py**

```
import sys
import time
import random

from sentence_indices import locations, clean_sentence
from hashing_approach import build_dictionary

def main():
    tests = int(sys.argv[1])
    print("doing {} tests".format(tests))
```

```
    words = clean_sentence(sys.stdin.read())
    print("read words")
    words_to_test = [random.choice(words) for _ in range(tests)]
    print("generated test words\nstarting naive test")
    naive_start = time.time()
    naive_locs = [locations(words, i) for i in words_to_test]
    naive_time = time.time() - naive_start
    print("finished naive test\nbeginning hash test")
    hash_start = time.time()
    loc_dict = build_dictionary(words)
    hash_dict_time = time.time() - hash_start
    hash_locs = [loc_dict[i] for i in words_to_test]
    hash_time = time.time() - hash_start
    print("they {}agree"
          .format("dis" if hash_locs != naive_locs else ""))
    print("naive time was {:.4f}".format(naive_time))
    print("hash time was  {:.4f}"
          " ({:.4f} of which spent building dictionary)"
          .format(hash_time, hash_dict_time))

if __name__ == "__main__":
    main()
```

# Task 2 - Compressing a file of capital letters and spaces

The specification of this task is to build a program that puts the individual words in a sentence in a list, replaces each word in the sentence with its location in the list, and then writes the list and the locations of the words to a file in a way that the original sentence can be reconstructed. This is effectively a form of compression, which is how I will be thinking of the task. This task also appears to be closely related to task #3, so in my approach to this task I will anticipate task #3 somewhat. Task #3 will be about compression, so I will approach this task as the lossy compression element of that (making assumptions about the form of the text file), and task #3 as the lossless approach. Because much of this task can be reused for task #3, I will also be writing decompressor scripts to accompany the compressor code I write in this task.

## Approach #1: readable_compression.py (a human-readable encoding)

Possibly the most naive way to approach this is to process all the words accordingly, write the words to a file, and then just to write the list of locations to the file as base 10 ASCII strings (this last part being the very naive behaviour). I will do this approach first, for

completeness. Note that this is not a waste of time, as very much of this approach can be reused for a more sophisticated approach.

For input, I will still be using STDIN (effectively a potentially very large file). This file will need to be transformed into a simple list of words, which will be done by:

## Algorithm #1: get_words (splitting a file into words)

Some more functionality that this algorithm can have is to strip this file of any punctuation, make lowercase letters uppercase, and use runs of whitespace as a delimiter. This would allow it to be tested on any text file, rather than only one meeting the requirements - and in fact would turn into some sort of lossy compression, as a decompressor algorithm could only reconstruct uppercase words separated by spaces.

Because this function needs to work with a file, it will be a little complicated to demonstrate use inside the interpreter. I will supply it with a dummy file, using a StringIO object. This is “an in-memory stream for text I/O.” I can initialise it with a string. Further complications come from the fact that I will implement the function as a generator (about which I will talk more later), but this means that I need to coerce its output into a list. Because of all this, I am going to use a helper function written later, under the Testing section of this approach.

Now here are some example cases:

```
>>> get_input_result(get_words, "One TwO THREE", [], wrapper=list)
['ONE', 'TWO', 'THREE']
>>> get_input_result(get_words, "there's 1 th1ng", [], wrapper=list)
['THERES', 'THNG']
```

And some edge cases:

```
>>> get_input_result(get_words, " tWo ", [], wrapper=list)
['TWO']
>>> get_input_result(get_words, "", [], wrapper=list)
[]
```

These are mostly with leading and trailing whitespace, or just whitespace.

The algorithm would work like this:

```
initialise an empty string to contain a word
for each character in the file:
    ignore the character if it is punctuation
    if the character is whitespace, and the word is not empty:
        "do something" with the word
        set the word to an empty string
    if the character is a letter:
        add the uppercased character to the word string
```

```
if the word is not empty:
    "do something" with the word
```

Note that I have been deliberately ambiguous about what to do when a complete word has been found. The word could be put in a list, and then have the list be returned, but for very large files this would read them all into memory, which might not be necessary. Also, the process of building a list and adding words and then returning it is quite cumbersome - and in this case, can't be simplified into a list comprehension, because the EOF has to be handled conditionally. Writing this function in Python comes out to the following:

```
def get_words(words_file: TextIO) -> Generator[str, None, None]:
    word: List[str] = []
    c: str = words_file.read(1)
    while c:
        if c in LETTERS:
            word.append(c.upper())
        elif c in WHITESPACE:
            if word:
                yield "".join(word)
                word = []
        c = words_file.read(1)
    if word:
        yield "".join(word)
```

What I have done here is used a generator function. This is a function that instead of using the return keyword to return a whole object, continually uses the yield keyword to yield multiple objects from the function, which can then be iterated over. This is similar to lazy evaluation in a language like Haskell, as the generator function will not actually run anything until the next object to iterate over is required. This is desirable when dealing with a large file like this, as it will not be unnecessarily read into memory.

Also, to accumulate the word, I am actually using a list of characters (which are strings, in Python), and later joining this. This is because a list is mutable, and can be appended to in linear time. The equivalent operation if I was to directly be accumulating a string would be word += c, which is notoriously bad for time complexity, as it performs an entire string concatenation every time, resulting in it running in quadratic time.

This function uses file.read(1) to read a single character from the given file, at a time. Note that it has therefore been written to take any file as parameter. This is using the idea of duck typing - if the object given to the function supports .read, (quacks) it is a good enough file (duck) for the function. This allows the function to be applied flexibly.

Another thing this function does, is when checking if the word is not empty, doing this by simply testing the truth value of the word. A more explicit way to test this would be to do something like

```
if len(word) != 0:
    ...
```

(Note that this is actually valid Python - … is the special Ellipse placeholder object.)

This explicitly determines the length of the string, and tests if this is not equal to 0, evaluating to a boolean. However, in Python, a condition does not have to be boolean. Python can evaluate the “truthiness” of an object used as a condition. The truthiness of an empty list [] is False, but the truthiness of any other list is True. This is exactly the behaviour I want.

## Algorithm #2: compress_words (converting words into unique words and locations)

This algorithm will take a series of words (which algorithm #1 has read for us), and transform these into a list of the unique words, and then a list which has for each word in the input substituted it for its location - I will call this my list of pointers. To preserve the lazy evaluation aspect of my code, this has to be put into one function, as both of its outputs need to keep track of each other.

This algorithm will also build a dictionary, to use for getting the location of a new word in the list of unique words to put into the list of pointers to each word.

Here are some example tests:

```
>>> compress_words(["ONE", "TWO", "ONE"])
(['ONE', 'TWO'], [0, 1, 0])
>>> compress_words(["FOO", "BAR"])
(['FOO', 'BAR'], [0, 1])
```

And some edge cases:

```
>>> compress_words(["ONE"])
(['ONE'], [0])
>>> compress_words([])
([], [])
```

In pseudocode, the algorithm looks like this:

```
initialise an empty dictionary to map words to their location
initialise an empty list to contain unique words
initialise a variable to track the number of unique words to 0
initialise an empty list to contain pointers
for each word in the input:
    if the word is already in the dictionary:
        lookup the word's position in the dictionary
        append this to the list of pointers
    else:
        push the word to the dictionary, mapping it to the current
number of unique words
        append the word to the list of unique words
        append the current number of unique words to the list of
pointers
        increment the current number of unique words by 1
```

Note that this algorithm uses 0-based indexing rather than 1-based. This makes more sense, as if we use 1-based we just lose the use of the number 0, so can encode one less pointer with a given number of characters.

Another thing to discuss is that this will return two things - a list of unique words used for encoding, and a list of pointers. In some languages, this might be a problem to try and return from a function. However, Python is pretty lax about return types - I can separate the values with a comma to return both. This is actually tuple constructor syntax and I am returning a tuple that looks like this (using Python's type hint syntax from the typing module): Tuple[List[str], List[int]]. Then, I can use tuple unpacking later when calling the function to unpack these lists, effectively without ever having to do much work in terms of constructing and deconstructing the tuple:

```
words_list, pointers = compress_words(...)
```

The algorithm now looks like so:

```
def compress_words(words: Iterable[str]) -> Tuple[List[str], List[int]]:
    words_dict: Dict[str, int] = {}
    words_list: List[str] = []
    words_size: int = 0
    pointers: List[int] = []
    word: str
    for word in words:
        if word in words_dict:
            pointers.append(words_dict[word])
        else:
            words_dict[word] = words_size
            words_list.append(word)
            pointers.append(words_size)
            words_size += 1
    return words_list, pointers
```

## Algorithm #3: write_dictionary (writing the unique words to a file)

Here I need to write a list of words to a file in a way that they can be read back into the same list used to encode the pointers. Seeing as they are all only composed of letters, an easy and intuitive way is to just separate them by a character like a space, and end the list with a newline. There is only a small amount of logic that the algorithm need to consider - if it has to write spaces between each word but not one at the end, it can't add a space for each item it iterates over (although there is a Pythonic way to deal with this). Note that this will not need to return anything, it just has the appropriate side-effect - writing the word to the file.

This algorithm is again a little complicated to demonstrate. Happily, I can use another helper function from the Testing section to do so - using this, I can demonstrate some expected outputs to a file:

```
>>> get_output_result(write_dictionary, [["FOO", "BAR"]])
"FOO BAR\n"
>>> get_output_result(write_dictionary, [["ONE", "TWO", "THREE"]])
"ONE TWO THREE\n"
```

And in some edge cases:

```
>>> get_output_result(write_dictionary, [["A"]])
"A\n"
>>> get_output_result(write_dictionary, [[]])
"\n"
```

The algorithm would look something like this:

```
set a variable that permits writing spaces to false
for each word to write:
    if writing a space is permitted:
        write a space
    else:
        permit writing spaces
    write the word
write a newline
```

However, if there is a method in Python to concatenate a list of strings together, using a certain string to go in between adjacent strings. This is str.join. One thing to note is that .join will not preserve the “laziness” of any code - if I used this with the result from get_words, it would read all the words into memory, which is not necessary. However, the list of unique words has already been read into memory, so I don’t need to worry about this. Something else that I can add is that the user can supply the separator and ending value to use, but I can have them as keyword arguments, which default to space and newline. This adds functionality but does not require the function caller to add any more arguments if they don’t want this functionality. Using all of this this, I can implement the algorithm like so in Python:

```
def write_dictionary(out_file: IO, words: List[str],
      separator: SomeText = " ", end: SomeText = "\n") -> None:
    out_file.write(separator.join(words) + end)
```

## Algorithm #4: write_pointers (writing the calculated integers to a file)

This algorithm will need to write a series of integers to a file, separated by spaces. It is the area of this approach that can be most improved upon. However, this time, I will just let it naively encode integers using base 10, ASCII digits from ‘0’ - ‘9’. This, similarly, will not need to return anything.

Just as with write_dictionary, I can use get_output_result to demonstrate the expected behaviour:

```
>>> get_output_result(write_pointers, [[1, 2, 3]])
"1 2 3"
>>> get_output_result(write_pointers, [[1, 0, 1]])
"1 0 1"
```

And in some edge cases:

```
>>> get_output_result(write_pointers, [[2]])
"2"
```

```
>>> get_output_result(write_pointers, [[]])
""
```

I can use pretty much the same algorithm as #3 - I only need to first convert the integers to strings. In Python, an integer can be converted to a readable base 10 string simply by calling str on the integer. I need to do this operation on every item in the list of pointers, so this can be done quickly using a list comprehension in Python -

```
[str(pointer) for pointer in pointers]
```

However, I don't actually need to build up this whole list in memory - I just need to be able to iterate over every produced string and write it to a file. This is like a generator function - but there is an easy way to build a generator out of a list comprehension. Changing the square [] brackets to () parentheses, turns it into a generator expression which behaves just like a generator function. Note that the parentheses do not have to be added if the generator expression is being constructed as a parameter to a function call. This means that inside str.join, I do not need to add double parentheses.

Also, as with #3, I have decided to allow the user to specify another encoding function for the pointer, and it defaults to the str constructor, and have done the same with separators.

```
def write_pointers(out_file: IO, pointers: List[int], encoder: Callable[[int],
      SomeText] = str, separator: SomeText = " ") -> None:
    pointer: int
    out_file.write(separator.join(encoder(pointer) for pointer in pointers))
```

## Class #1 (std_streams):

This is a class to contain two file objects, and handle their closing after passing to a context manager - this is for the slight convenience of being able to return a std_streams object and then pass this to a with statement, and also improves code clarity. The code doesn't do much - all it needs to do is store, return and close the file objects given.

This class implements a few methods. It implements __init__, the constructor function, which quite simply takes two files and assigns the files to attributes of the object. The __enter__ method is called by the with statement when it is entered. No special preparation of the files needs to occur, so it can just return the two files. The __exit__ method is called when a with statement exits. This needs to close the files. It also takes a whole bunch of arguments about the status of the exit, but this does not need to be used, at all, it only needs to worry about closing the files. Lastly I have implemented the __repr__ method, defining how the string representation of the object should be obtained. This is so I can demonstrate the usage of get_std_streams.

```
class std_streams:
    def __init__(self, stdin: IO, stdout: IO) -> None:
        self.stdin: IO = stdin
        self.stdout: IO = stdout

    def __enter__(self) -> Tuple[IO, IO]:
```

```
        return self.stdin, self.stdout

    def __exit__(self, error_type: type, value: Exception,
                 traceback: "Traceback object") -> None:
        self.stdin.close()
        self.stdout.close()

    def __repr__(self) -> str:
        return "std_streams({}, {})".format(self.stdin, self.stdout)
```

## Subroutine #1 get_std_streams (parsing command line arguments to obtain the files to read from and write to):

There is however still more to come inside of this approach. All of the theoretical programming is done, but I still need some more administrative functionalities, which I am calling subroutines.

Because I know I will have to work with files eventually, I have decided to get that out of the way now. I am going to write a function that will provide all the logic for determining files to read from and write to, or alternatively, which file like objects should be used like the standard streams, hence why it is called get_std_streams.

To find this I am parsing given command line arguments, using Python's argparse module. I will make it so that the user can either use the flag --input or --output to specify a filename, or it will default to stdin or stdout. argparse will really do most of the thinking, here - I only provide it with a few parameters and defaults.

In anticipation of my later approaches, I have also given the function the functionality of providing binary files. This is handled by a few ternary operators. Binary files are opened with "rb" or "wb" instead of "r" or "w" as file mode, and in the case of stdin and stdout, the underlying binary buffer can be accessed with sys.stdin.buffer and sys.stdin.stdout.

Because this isn't an absolute argument parser, but it just parses the arguments relevant to it, it then reassigns the remaining arguments to the list given. It does this by assigning to the slice [:] of the list given, rather than assigning to the list. This is because if I was to assign to the name argv, this would be an operation handled by Python's variable dictionary, and as I was making an assignment to a name inside of a scope (function) it would make a new local variable and not affect the actual list. Instead, when I assign to the slice [:] I am actually calling the __setitem__ method of the list, so am changing the value of the actual list in memory.

The function returns a std_streams object, which is a class that I have defined myself, which, as mentioned earlier, can contain and handle the closing of two file objects.

```
def get_std_streams(argv: List[str], in_binary: bool = False,
                    out_binary: bool = False) -> std_streams:
    parser: argparse.ArgumentParser = argparse.ArgumentParser()
    parser.add_argument("--input", type=argparse.FileType('rb' if in_binary else
                        'r'), default=(sys.stdin.buffer if in_binary else sys.stdin))
    parser.add_argument("--output", type=argparse.FileType('wb' if out_binary else
```

```
                    'w'), default=(sys.stdout.buffer if out_binary else
sys.stdout))
    args: argparse.Namespace
    remaining: List[str]
    args, remaining = parser.parse_known_args(argv)
    argv[:] = remaining
    return std_streams(args.input, args.output)
```

## Main

The main function now needs to do relatively little - it calls all of the functions and links them up appropriately, supplying STDIN and STDOUT as input and output files.

```
def main(stdin: TextIO, stdout: TextIO, argv: List[str]) -> None:
    words: List[str] = get_words(stdin)
    unique_words: List[str]
    pointers: List[int]
    unique_words, pointers = compress_words(words)
    write_dictionary(stdout, unique_words, " ", "\n")
    write_pointers(stdout, pointers)
```

It can now be called on our original example like so:

```
$ echo "ASK NOT WHAT YouR countRY CAN do for you ASK WHAT YOU CAN DO
FOR YOUR COUNTRY" | python readable_compression.py

ASK NOT WHAT YOUR COUNTRY CAN DO FOR YOU
0 1 2 3 4 5 6 7 8 0 2 8 5 6 7 3 4
```

We can see that it behaves correctly - the only difference is that all of the indices are one lower. If we add the word ‘but’ to the sentence, we can see that this makes a difference - where my script would produce the number 9, the example would produce 10, using an extra character.

```
$ echo "ASK NOT WHAT YouR countRY CAN do for you but ASK WHAT YOU CAN
DO FOR YOUR COUNTRY" | python readable_compression.py

ASK NOT WHAT YOUR COUNTRY CAN DO FOR YOU BUT
0 1 2 3 4 5 6 7 8 9 2 8 5 6 7 3 4
```

Note that I this now works with files as input or output as well. If the file input.txt contains the following:

```
ASK NOT WHAT YouR countRY CAN do for you but ASK WHAT YOU CAN DO
FOR YOUR COUNTRY
```

And I call

```
$ python readable_compression.py --input input.txt --output compressed
```

That same output of

```
ASK NOT WHAT YOUR COUNTRY CAN DO FOR YOU BUT
0 1 2 3 4 5 6 7 8 9 2 8 5 6 7 3 4
```

Is written to the file “compressed”. This will serve as my demonstration that it works with files, as nothing is inherently different about any of my methods that work with files, whether they’re writing to stdout or another output file doesn’t change any behaviour. I will be using stdin and stdout as my main forms of input and output for the purposes of demonstration from now on, as it is clearer to fit in one command.

**#readable_compression.py**

```
"""
A script to perform a simplistic lossy compression on text, by creating an index
of unique words and using references to this index to store words. These
"pointers" are encoded in normal human readable base 10 numbers.
"""

import sys
import string
import argparse

from typing import *
from typing.io import *

SomeText = Union[str, bytes]

WHITESPACE: Set[str] = set(string.whitespace)
LETTERS: Set[str] = set(string.ascii_letters)

class std_streams:
    """
    An object to wrap around two file like objects, and provide boilerplate
    methods to release these files to a with statement, and clean up afterwards.
    """

    def __init__(self, stdin: IO, stdout: IO) -> None:
        self.stdin: IO = stdin
        self.stdout: IO = stdout

    def __enter__(self) -> Tuple[IO, IO]:
       return self.stdin, self.stdout

    def __exit__(self, error_type: type, value: Exception,
          traceback: "Traceback object") -> None:
        self.stdin.close()
        self.stdout.close()

    def __repr__(self) -> str:
       return "std_streams({}, {})".format(self.stdin, self.stdout)

def get_std_streams(argv: List[str], in_binary: bool = False,
      out_binary: bool = False) -> std_streams:
    """
    Get an input to read from and an output to write to, by parsing the given
    arguments, using --input and --output flags, defaulting to stdin and stdout.
    Allows binary mode to be requested.

    Example usage:
    >>> get_std_streams(["--output", "out.txt"], in_binary=True)
    std_streams(<_io.BufferedReader name='<stdin>'>,
                <_io.TextIOWrapper name='out.txt' mode='w' encoding='UTF-8'>)
```

```
    >>> get_std_streams(["--foo", "1", "3"])
    std_streams(<_io.TextIOWrapper name='<stdin>' mode='r' encoding='UTF-8'>,
                <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>)
    >>> get_std_streams([])
    std_streams(<_io.TextIOWrapper name='<stdin>' mode='r' encoding='UTF-8'>,
                <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>)
    >>> get_std_streams(["update"], in_binary=True, out_binary=True)
    std_streams(<_io.BufferedReader name='<stdin>'>,
                <_io.BufferedWriter name='<stdout>'>)

    Parameters:
    argv - List[str] - a list of strings, as might be found in sys.argv,
     representing the command line arguments to be used (and modified)
    in_binary (optional keyword argument) - bool - boolean representing whether
     the input should be in binary mode. Defaults to False
    out_binary (optional keyword argument) - bool - boolean representing whether
     the output should be in binary mode. Defaults to False

    Return:
    std_streams - a std_streams object - wraps files to be given to a context
     manager.  see std_streams documentation
    """
   parser: argparse.ArgumentParser = argparse.ArgumentParser()
    parser.add_argument("--input", type=argparse.FileType('rb' if in_binary else
                    'r'), default=(sys.stdin.buffer if in_binary else sys.stdin))
    parser.add_argument("--output", type=argparse.FileType('wb' if out_binary else
                    'w'), default=(sys.stdout.buffer if out_binary else
sys.stdout))
    args: argparse.Namespace
    remaining: List[str]
    args, remaining = parser.parse_known_args(argv)
    argv[:] = remaining
    return std_streams(args.input, args.output)

def get_words(words_file: TextIO) -> Generator[str, None, None]:
    """
    Read whitespace separated words from a file. Ignores punctuation.

    Example usage (with the get_input_result helper function from
     test_readable_compression):
    >>> get_input_result(get_words, "One TwO THREE", [], wrapper=list)
    ['ONE', 'TWO', 'THREE']
    >>> get_input_result(get_words, "there's 1 th1ng", [], wrapper=list)
    ['THERES', 'THNG']
    >>> get_input_result(get_words, " tWo ", [], wrapper=list)
    ['TWO']
    >>> get_input_result(get_words, "", [], wrapper=list)
    []

    Parameters:
    words_file - TextIO - a text file to read strings fromm

    Return:
    Generator[str, None, None] - a generator of strings, representing words
     extracted from the file

    """

    word: List[str] = []
    c: str = words_file.read(1)
    while c:
        if c in LETTERS:
            word.append(c.upper())
        elif c in WHITESPACE:
            if word:
                yield "".join(word)
```

```
                word = []
            c = words_file.read(1)
        if word:
            yield "".join(word)

def compress_words(words: Iterable[str]) -> Tuple[List[str], List[int]]:
    """
    Compresses a series of words into a list of unique words and a list of
    indices for each word in the unique list.

    Example usage:
    >>> compress_words(["ONE", "TWO", "ONE"])
    (['ONE', 'TWO'], [0, 1, 0])
    >>> compress_words(["FOO", "BAR"])
    (['FOO', 'BAR'], [0, 1])
    >>> compress_words([])
    ([], [])

    Parameters:
    words - Iterable[str] - an iterable of strings, made of uppercase ascii letters

    Return:
    Tuple[List[str], List[int]] - a tuple of a list of the unique words and a list
     of words indices

    """

    words_dict: Dict[str, int] = {}
    words_list: List[str] = []
    words_size: int = 0
    pointers: List[int] = []
    word: str
    for word in words:
        if word in words_dict:
            pointers.append(words_dict[word])
        else:
            words_dict[word] = words_size
            words_list.append(word)
            pointers.append(words_size)
            words_size += 1
    return words_list, pointers

def write_dictionary(out_file: IO, words: List[str],
      separator: SomeText = " ", end: SomeText = "\n") -> None:
   """
    Writes unique words to a file, followed by a newline

    Example usage (with get_output_result):
    >>> get_output_result(write_dictionary, [["FOO", "BAR"]])
    "FOO BAR\n"
    >>> get_output_result(write_dictionary, [["A"]])
    "A\n"
    >>> get_output_result(write_dictionary, [[]])
    "\n"

    Parameters:
    out_file - TextIO - an output text file to write to
    words - List[str] - the list of unique words to write to file
    separator - SomeText - the the separator between words. defaults to space
    end - SomeText - the final, ending separator string to write. defaults to
     newline

    Return:
    None
    """
```

```
    out_file.write(separator.join(words) + end)

def write_pointers(out_file: TextIO, pointers: List[int], encoder: Callable[[int],
      SomeText] = str, separator: SomeText = " ") -> None:
    """
    Writes pointers to a file

    Example usage:
    >>> get_output_result(write_pointers, [[1, 2, 3]])
    "1 2 3"
    >>> get_output_result(write_pointers, [[2]])
    "2"
    >>> get_output_result(write_pointers, [[]])
    ""

    Parameters:
    out_file - TextIO - an output text file to write to
    words - List[int] - the list of pointers to write to file
    encoder - Callable[[int], Sometext] - function that encoded a an integer value
     of a pointer to SomeText

    Return:
    None
    """

    pointer: int
    out_file.write(separator.join(encoder(pointer) for pointer in pointers))

def main(stdin: TextIO, stdout: TextIO, argv: List[str]) -> None:
    words: List[str] = get_words(stdin)
    unique_words: List[str]
    pointers: List[int]
    unique_words, pointers = compress_words(words)
    write_dictionary(stdout, unique_words, " ", "\n")
    write_pointers(stdout, pointers)

if __name__ == "__main__":
    stdin: TextIO
    stdout: TextIO
    with get_std_streams(sys.argv) as (stdin, stdout):
        main(stdin, stdout, sys.argv)
```

## Testing

Here I write two important functions: get_output_result and get_input_result. In the interest of time, I won't talk about how they work much but their documentation should provide some information.

**#test_readable_compression.py**

```
import unittest

from io import StringIO, BytesIO

from readable_compression import *

from typing import *

def get_output_result(func: Callable[..., None], args: Iterable[Any],
      binary: bool = False) -> Union[str, bytes]:
```

```
    """
    A function to simulate an output file for another function, using a StringIO
    or BytesIO object, and return the contents of the file after processing. It
    assumes the file object is the first argument to the function, and it can take
    remaining arguments to be passed.

    Example usage:
    >>> def f(out_file):
    ...     out_file.write("foo")
    >>> get_output_result(f, [])
    "foo"
    >>> def g(out_file, n):
    ...     out_file.write(str(n + 1))
    >>> get_output_result(g, [2])
    "3"

    Parameters:
    func - Callable[..., None] - function which will process the file
    args - Iterable[Any] - remaining arguments for func
    binary - bool - boolean determining whether the file should be binary, ie
     whether BytesIO or StringIO should be used

    Return:
    Union[str, bytes] - The contents of the processed file. either str or bytes
     depending on the value of binary
    """

    output: IO = BytesIO() if binary else StringIO()
    func(output, *args)
    return output.getvalue()

def get_input_result(func: Callable[..., Any], in_val: Union[str, bytes], args:
      Iterable[Any], wrapper: Callable[[Any], Any] = lambda x: x,
      binary: bool = False) -> Any:
    """
    A function to simulate an input file for another function, using a StringIO or
    BytesIO object, and return the return value of the function, allowing for this
    to be passed through a wrapper function. It assumes the file object is the
    first argument to the function, and it can take remaining arguments to be
    passed.

    Example usage (these are somewhat contrived):
    >>> def f(in_file):
    ...     return in_file.read() + "bar"
    >>> get_input_result(f, "foo", [])
    "foobar"
    >>> def g(binary_file, increment):
    ...     return [ord(binary_file.read(1)) + increment]
    >>> get_input_result(g, b"\x80", [2], binary=True, wrapper=bytes)
    b"\x82"

    Parameters:
    func - Callable[..., Any] - the function which will take input from the file
    in_val - Union[str, bytes] - the contents of the file. Type depends on the
     value of binary
    args - Iterable[Any] - remaining arguments to pass to the file
    wrapper - Callable[[Any], [Any]] - optional wrapper function, defaults to
     an identity operator
    binary - bool - boolean signifying whether it is a binary or text file.
     defaults to False

    Return:
    Any - the return value of the wrapper function applied to the return value
     of the function
    """
```

```
    in_file: IO = (BytesIO if binary else StringIO)(in_val)
    return wrapper(func(in_file, *args))

class TestReadableCompression(unittest.TestCase):
    def test_get_words(self) -> None:
        self.assertEqual(get_input_result(get_words, "One TwO THREE", [],
                                 wrapper=list), ["ONE", "TWO", "THREE"])
        self.assertEqual(get_input_result(get_words, "", [], wrapper=list), [])
        self.assertEqual(get_input_result(get_words, "   ", [], wrapper=list), [])
        self.assertEqual(get_input_result(get_words, " TwO ", [], wrapper=list),
                         ["TWO"])
        self.assertEqual(get_input_result(get_words, "there's 1 th1ng", [],
                                 wrapper=list), ["THERES", "THNG"])

    def test_compress_words(self) -> None:
        self.assertEqual(compress_words(["ONE", "TWO", "ONE"]), (["ONE", "TWO"],
                         [0, 1, 0]))
        self.assertEqual(compress_words(["ONE"]), (["ONE"], [0]))
        self.assertEqual(compress_words([]), ([], []))

    def test_write_dictionary(self) -> None:
        self.assertEqual(get_output_result(write_dictionary, [["FOO", "BAR"],
                         " ", "\n"]), "FOO BAR\n")
        self.assertEqual(get_output_result(write_dictionary, [["A"],
                         " ", "\n"]), "A\n")
        self.assertEqual(get_output_result(write_dictionary, [[],
                         " ", "\n"]), "\n")

    def test_write_pointers(self) -> None:
        self.assertEqual(get_output_result(write_pointers, [[1, 2, 3]]), "1 2 3")
        self.assertEqual(get_output_result(write_pointers, [[2]]), "2")
        self.assertEqual(get_output_result(write_pointers, [[]]), "")
```

# Approach #1b: readable_decompression.py (the accompanying decompressor)

This will need to decompress the file produced by the previous code. It will need to be able to read back the list of unique words at the start of the previous file, read the pointers in the file, and bring those together to reconstruct the original file. These are the three algorithms required for this task.

## Algorithm #1: read_dictionary (reading back the list of unique words)

This algorithm will need to continue to read words from the file, separated by spaces, until it encounters a newline, at which point it can stop. These words will then be used to decode the pointers later in the file.

Here is some example usage:

```
>>> get_input_result(read_dictionary, "ONE TWO THREE\nabc", [],
wrapper=list)
['ONE', 'TWO', 'THREE']
```

```
>>> get_input_result(read_dictionary, "ONE\n", [], wrapper=list)
['ONE']
>>> get_input_result(read_dictionary, "FOO BAR\n", [], wrapper=list)
['FOO', 'BAR']
```

For mapping words to locations, I used a dictionary, but there is a better data structure for mapping contiguous integers to any object - the list (or array), of course! It is easy to use the generator syntax and build a list out of the generator using Python list() constructor. In pseudocode, the algorithm can be done as follows:

```
initialise an empty string to contain a word
for each character until this character is a newline:
    if the character is a space:
        yield the current word
        set the word to an empty string
    else:
        append the character to the word
yield the last word
```

In Python this can be achieved with the following code:

```
def read_dictionary(in_file: TextIO, separator: SomeText = " ",
      end: SomeText = "\n") -> Generator[str, None, None]:
    c: str = in_file.read(1)
    word: List[str] = []
    while c != end:
        if c == separator:
            yield "".join(word)
            word = []
        else:
            word.append(c)
        c = in_file.read(1)
    if word:
        yield "".join(word)
```

## Algorithm #2: read_pointers (reading pointers from file)

This algorithm will, for the remainder of the file, read space separated pointers from the file. I will implement it as a generator as it isn't necessary for all the pointers to be in memory.

Now here is its expected behaviour:

```
>>> get_input_result(read_pointers, "3 2 1", [], wrapper=list)
[3, 2, 1]
>>> get_input_result(read_pointers, "2", [], wrapper=list)
[2]
>>> get_input_result(read_pointers, "4 3", [], wrapper=list)
[4, 3]
```

In pseudocode, this algorithm can be implemented like this:

```
initialise an empty string to contain the current pointer
for each character in the file:
    if the character is a space:
        parse the current word into an integer
        yield this integer
    else:
        append the character to the pointer-string
yield the last pointer (as an integer)
```

This can then be done, in Python, like so:

```
def read_pointers(in_file: TextIO, decoder: Callable[[SomeText], int] = int,
      separator: SomeText = " ") -> Generator[int, None, None]:
    c: str = in_file.read(1)
    n: List[str] = []
    while c:
        if c == separator:
            yield decoder("".join(n))
            n = []
        else:
            n.append(c)
        c = in_file.read(1)
    if n:
        yield decoder("".join(n))
```

## Algorithm #3: decompress (uses the list of unique words and the list of pointers to reconstruct the original file

This is quite a simple algorithm - it needs to iterate over the pointers and output the corresponding word for each pointer, separated by spaces. Here is some expected behaviour:

```
>>> get_output_result(decompress, [[1, 0], ["FOO", "BAR"]])
'BAR FOO'
>>> get_output_result(decompress, [[0], ["FOO"]])
'FOO'
>>> get_output_result(decompress, [[], []])
''
```

Last time I needed to join a number of strings with a space, I could use .join, as all the words were already stored in memory. However, to preserve the laziness of my code here, I will need to explicitly write each space, with a small amount of logic to stop me from writing an extra space. Apart from this, and a little bit of logic to obtain each word, this function is very similar to readable_compression.write_dictionary, in pseudocode:

```
set a variable that permits writing spaces to false
for each pointer:
    if writing a space is permitted:
        write a space
    else:
        permit writing spaces
    find the word to be written at the index in the list of unique
words with the value of the pointer
    write this word
```

In Python, this can be done like so:

```
def decompress(out_file: TextIO, pointers: Iterable[int], words: List[str]) ->
None:
    """
    Decompress a series of pointer values, using a list of words, and write them
    to an output file.

    Parameters:
    out_file - TextIO - file to write to
    pointers - Iterable[int] - pointers to decompress
    words - List[str] - index of words to decompress them with

    Return:
    None
    """

    do_space: bool = False
    i: int
    for i in pointers:
        if do_space:
            out_file.write(" ")
        else:
            do_space = True
        out_file.write(words[i])
```

## Main

Again, because my functions do so much, all the main function has to do is link them up, and supply STDIN and STDOUT as the input and output files. It coerces the generator of word to a list. It also has to be careful to read things in the same order that they were written by readable_compression (ie unique words then pointers).

```
def main(stdin: TextIO, stdout: TextIO, argv: List[str]) -> None:
    words: List[str] = list(read_dictionary(stdin, " ", "\n"))
    pointers: Generator[int, None, None] = read_pointers(stdin, int, " ")
    decompress(stdout, pointers, words)
```

Now, by using this program with the output of readable_compression.py, we can see that it works. To also test the capability to filter punctuation and capitalise, I have added a comma and several lowercase letters.

```
$ echo "ASK Not what your country CAN do FoR yOu country, ask WHAT YOU
caN Do FOR YOUR COUNTRY" | python readable_compression.py | python
readable_decompression.py
```

Output:

```
ASK NOT WHAT YOUR COUNTRY CAN DO FOR YOU COUNTRY ASK WHAT YOU CAN DO
FOR YOUR COUNTRY
```

We can see that it works!

**#readable_decompression.py**

```
"""
A script to decompress output from readable_compression back into normal
(capitalised) text
"""

import sys

from readable_compression import get_std_streams, SomeText

from typing import *
from typing.io import *

def read_dictionary(in_file: TextIO, separator: SomeText = " ",
      end: SomeText = "\n") -> Generator[str, None, None]:
    """
    Read the list of unique words used as an encoding dictionary from the start of
    the compressed file, given a separator value and a value to end on.

    Example usage (with get_input_result helper):
    >>> get_input_result(read_dictionary, "ONE TWO THREE\nabc", [], wrapper=list)
    ['ONE', 'TWO', 'THREE']
    >>> get_input_result(read_dictionary, "ONE\n", [], wrapper=list)
    ['ONE']
    >>> get_input_result(read_dictionary, "FOO BAR\n", [], wrapper=list)
    ['FOO', 'BAR']

    Parameters:
    in_file = TextIO - file to read from
    separator - sometext - text that separates words to read. defaults to space
    end - SomeText - text htat marks the ends of words to read. defaults to newline

    Return:
    Generator[str, None, None] - a generator of word read from the file
    """


    c: str = in_file.read(1)
    word: List[str] = []
    while c != end:
        if c == separator:
            yield "".join(word)
            word = []
        else:
            word.append(c)
        c = in_file.read(1)
    if word:
        yield "".join(word)

def read_pointers(in_file: TextIO, decoder: Callable[[SomeText], int] = int,
      separator: SomeText = " ") -> Generator[int, None, None]:
    """
    Read and decode pointers from compressed file, given a separator value and
    a decoder function, which defaults to the base 10 int constructor where all
    the unique words have been consumed. Reads until EOF.
```

```
    Example usage:
    >>> get_input_result(read_pointers, "3 2 1", [], wrapper=list)
    [3, 2, 1]
    >>> get_input_result(read_pointers, "2", [], wrapper=list)
    [2]
    >>> get_input_result(read_pointers, "4 3", [], wrapper=list)
    [4, 3]

    Parameters:
    in_file - TextIO - file to be read from
    decoder - Callable[[SomeText], int] - decoder function to be called on each
     pointer. defaults to int constructor
    separator - SomeText - separator text between pointers. defaults to space

    Return:
    Generator[int, None, None]
    """

    c: str = in_file.read(1)
    n: List[str] = []
    while c:
        if c == separator:
            yield decoder("".join(n))
            n = []
        else:
            n.append(c)
        c = in_file.read(1)
    if n:
        yield decoder("".join(n))

def decompress(out_file: TextIO, pointers: Iterable[int], words: List[str]) ->
None:
    """
    Using pointers and unique words, decompress the file and write this to a given
    output file

    Example usage:
    >>> get_output_result(decompress, [[1, 0], ["FOO", "BAR"]])
    'BAR FOO'
    >>> get_output_result(decompress, [[0], ["FOO"]])
    'FOO'
    >>> get_output_result(decompress, [[], []])
    ''

    Parameters:
    out_file - TextIO - text file to write to
    pointers - Iterable[int] - pointers to words to iterate over
    words - List[str] - words to index with pointers

    Return:
    None
    """

    do_space: bool = False
    i: int
    for i in pointers:
        if do_space:
            out_file.write(" ")
        else:
            do_space = True
        out_file.write(words[i])

def main(stdin: TextIO, stdout: TextIO, argv: List[str]) -> None:
    words: List[str] = list(read_dictionary(stdin, " ", "\n"))
    pointers: Generator[int, None, None] = read_pointers(stdin, int, " ")
```

```
    decompress(stdout, pointers, words)

if __name__ == "__main__":
    stdin: TextIO
    stdout: TextIO
    with get_std_streams(sys.argv) as (stdin, stdout):
        main(stdin, stdout, sys.argv)
```

## Testing

**#test_readable_decompression.py**

```
import unittest

from test_readable_compression import get_input_result, get_output_result

from readable_decompression import *

class TestReadableDecompression(unittest.TestCase):
    def test_read_dictionary(self) -> None:
        self.assertEqual(get_input_result(read_dictionary, "ONE TWO THREE\nabc",
                         [], wrapper=list), ["ONE", "TWO", "THREE"])
        self.assertEqual(get_input_result(read_dictionary, "ONE\n", [],
                         wrapper=list), ["ONE"])
        self.assertEqual(get_input_result(read_dictionary, "FOO BAR\n",
                         [], wrapper=list), ["FOO", "BAR"])

    def test_read_pointers(self) -> None:
        self.assertEqual(get_input_result(read_pointers, "3 2 1", [],
                         wrapper=list), [3, 2, 1])
        self.assertEqual(get_input_result(read_pointers, "2", [],
                         wrapper=list), [2])
        self.assertEqual(get_input_result(read_pointers, "4 3", [],
                         wrapper=list), [4, 3])

    def test_decompress(self) -> None:
        self.assertEqual(get_output_result(decompress, [[1, 0], ["FOO", "BAR"]]),
                         "BAR FOO")
        self.assertEqual(get_output_result(decompress, [[0], ["FOO"]]), "FOO")
        self.assertEqual(get_output_result(decompress, [[], []]), "")
```

# Approach #2: sorted_compression.py (sorting the unique words so more frequent words have a shorted code)

The first obvious improvement is that a word that occurs more often should have a shorter pointer representation. In this case, it should be as early in the list as possible, so as an integer it can be represented with as few digits as possible. For example, if the word “the” occurs 400 times, but is at index 10, this will take 800 digits after encoding as pointers. If it was moved to index 9, it would only take 400 digits, so would achieve a lot more compression. This algorithm only needs slight modification from readable_decompression - the generation of the unique list of words. This approach will not need a decompressor as it can be decompressed by readable_decompression.py.

## Algorithm #1: compile_dictionary (compile a sorted list of unique words)

This function will need to take a list of words and return a list of the uniquely appearing words sorted by their frequencies. I will need to count the number of instances of each word, and then sort by this. Here is some example behaviour:

```
>>> compile_dictionary(["FOO", "BAR", "FOO", "BAR", "FOO", "EGGS"])
({'FOO': 0, 'BAR': 1, 'EGGS': 2}, ['FOO', 'BAR', 'EGGS'])
>>> compile_dictionary(["FOO"])
({'FOO': 0}, ['FOO'])
>>> compile_dictionary([])
({}, [])
```

Because I need to know information about every word before I can be sure my list of words is in the right order, I can no longer preserve the laziness in this part of the code.

In pseudocode, the algorithm might look something like this:

```
>>> compile_dictionary(["FOO", "BAR", "FOO"])
({'BAR': 1, 'FOO': 0}, ['FOO', 'BAR'])
>>> compile_dictionary(["FOO"])
({'FOO': 0}, ['FOO'])
>>> compile_dictionary(["TWO", "TWO", "THREE", "THREE", "THREE"])
({'TWO': 1, 'THREE': 0}, ['THREE', 'TWO'])
>>> compile_dictionary([])
({}, [])
```

```
initialise a dictionary to tally the frequency of each word
for each word:
    increment the tally for that word
sort the list of words by their tallies
```

I won't need to go into depth on how the sorting works, as this is a builtin function provided by Python. However, this entire algorithm is also already provided in the Python standard library, as collections.Counter, a collection type designed to "support convenient and rapid tallies". Using this, I can implement the algorithm like so in Python:

```
def compile_dictionary(words: Iterable[str]) -> Tuple[Dict[str, int], List[str]]:
    c: collections.Counter = collections.Counter(words)
    words_list: List[str] = [i[0] for i in c.most_common()]
    i: str
    ind: int
    return {i: ind for ind, i in enumerate(words_list)}, words_list
```

This creates a Counter for words, and then uses the most_common() method of a counter, without any arguments, to get a list of (word, frequency) tuples, and uses a list comprehension to just pull the words from this list. It then builds a dictionary using a

dictionary comprehension. It returns, in the same fashion as readable_compression.get_words, a tuple, consisting of a dictionary to use for encoding, and a list of unique words to write to a file.

## Algorithm #2: compile_pointers (calculating the pointer for each word)

This algorithm is much like readable_compression.compress_words, but it has less work to do as the dictionary has already been built by compile_dictionary. In fact, it only has to look up each word in the dictionary, and then it is done. Here is some expected behaviour:

```
>>> list(compile_pointers(["A", "B", "C"], {"A": 1, "B": 0, "C": 2}))
[1, 0, 2]
>>> list(compile_pointers(["A", "B", "A"], {"A": 1, "B": 0}))
[1, 0, 1]
>>> list(compile_pointers(["A"], {"A": 1}))
[1]
>>> list(compile_pointers([], {}))
[]
```

In pseudocode:

```
for each word:
    look up the word in the dictionary
    yield the looked up pointer
```

Because all it does is one operation on each word, it can be implemented with a generator expression:

```
def compile_pointers(words: List[str],
      words_dict: Dict[str, int]) -> Generator[int, None, None]:
    word: str
    return (words_dict[word] for word in words)
```

## Main

This function is very similar to readable_compression.main. There are some differences, for instance, that it reads the entire list of words into memory. This is because it needs to be consumed twice - once for building the dictionary, and once for computing all the pointers. This means I have to read the generator from get_words into a list, using the list constructor.

```
def main(stdin: TextIO, stdout: TextIO, argv: List[str]) -> None:
    words: List[str] = list(get_words(stdin))
    words_dict: Dict[str, int]
    keywords: List[str]
    words_dict, keywords = compile_dictionary(words)
    pointers: Generator[int, None, None] = compile_pointers(words, words_dict)
    write_dictionary(stdout, keywords)
    write_pointers(stdout, pointers)
```

I can now try out this script for compression. I'm using a slightly modified input, to demonstrate that it sorts by frequency.

```
$ echo "ASK NOT WHAT YOUR COUNTRY CAN DO FOR YOU BUT YOU SHOULD ASK
WHAT YOU CAN DO FOR YOUR COUNTRY" | python sorted_compression.py

YOU CAN ASK COUNTRY DO FOR WHAT YOUR NOT SHOULD BUT
2 8 6 7 3 1 4 5 0 10 0 9 2 6 0 1 4 5 7 3
```

Now I can test it with an input designed specifically to be problematic for readable_compression.py, with a word that occurs late in the input, but occurs often, causing characters to be wasted.

```
$ echo "TWENTY THIRTY FORTY TEN NINE EIGHT SEVEN SIX FIVE FOUR FIFTEEN
FIFTEEN FIFTEEN ONE ONE ONE TWENTY ONE" | python
readable_compression.py

TWENTY THIRTY FORTY TEN NINE EIGHT SEVEN SIX FIVE FOUR FIFTEEN ONE
0 1 2 3 4 5 6 7 8 9 10 10 10 11 11 11 0 11
```

Now using sorted_compression.py, we can see that it is achieving more compression.

```
$ echo "TWENTY THIRTY FORTY TEN NINE EIGHT SEVEN SIX FIVE FOUR FIFTEEN
FIFTEEN FIFTEEN ONE ONE ONE TWENTY ONE" | python sorted_compression.py

ONE FIFTEEN TWENTY EIGHT TEN SEVEN FOUR THIRTY FIVE SIX FORTY NINE
2 7 10 4 11 3 5 9 8 6 1 1 1 0 0 0 2 0
```

To quickly compare, I can use the wc (word count) command, with -c parameter to measure byte count instead of words, to measure the size of the output.

Note that in Powershell, wc -c must be replicated with

```
Measure-Object -Character
```

```
$ echo "TWENTY THIRTY FORTY TEN NINE EIGHT SEVEN SIX FIVE FOUR FIFTEEN
FIFTEEN FIFTEEN ONE ONE ONE TWENTY ONE" | python sorted_compression.py
| wc -c

103
```

```
$ echo "TWENTY THIRTY FORTY TEN NINE EIGHT SEVEN SIX FIVE FOUR FIFTEEN
FIFTEEN FIFTEEN ONE ONE ONE TWENTY ONE" | python
readable_compression.py | wc -c

108
```

**#sorted_compression.py**

```
"""
A script to perform lossy compression similar to readable_compression.py, but
this sorts the word by frequency, so the most frequent words have the lowst
pointer values and therefore can be encoded in less space.
"""

import sys
import collections

from readable_compression import (get_std_streams, get_words,
                                  write_dictionary, write_pointers)

from typing import *
from typing.io import *

def compile_dictionary(words: Iterable[str]) -> Tuple[Dict[str, int], List[str]]:
    """
    Compile a dictionary and list of unique words sorted by frequency, using
    collections.Counter

    Example usage:
    >>> compile_dictionary(["FOO", "BAR", "FOO", "BAR", "FOO", "EGGS"])
    ({'FOO': 0, 'BAR': 1, 'EGGS': 2}, ['FOO', 'BAR', 'EGGS'])
    >>> compile_dictionary(["FOO"])
    ({'FOO': 0}, ['FOO'])
    >>> compile_dictionary([])
    ({}, [])

    Parameters:
    words - Iterable[str] - the sequence of words to process. these should all
     be uppercase sequences of ascii letters

    Return:
    Tuple[Dict[str, int], List[str]] - a tuple of the dictionary to be used in
     compressing, and of the list of unique words in order to write to file.
    """

    c: collections.Counter = collections.Counter(words)
    words_list: List[str] = [i[0] for i in c.most_common()]
    i: str
    ind: int
    return {i: ind for ind, i in enumerate(words_list)}, words_list

def compile_pointers(words: List[str],
      words_dict: Dict[str, int]) -> Generator[int, None, None]:
    """
    Using a dictionary of words to pointers, and a list of words, convert these
    words to pointers. A relatively simple function.

    Example usage:
    >>> compile_pointers(["FOO", "BAR"], {"FOO": 1, "BAR": 0})
    [1, 0]
    >>> compile_pointers(["ONE", "TWO", "ONE"], {"ONE": 0, "TWO": 1})
    [0, 1, 0]
    >>> compile_pointers([], {})
    []

    Parameters:
    words - List[str] - the list of words to process. these should all be
     uppercase
    words_dict - Dict[str, int] - the dictionary of words to pointers

    Return:
    Generator[int, None, None] - a generator of pointers
    """
```

```
    word: str
    return (words_dict[word] for word in words)

def main(stdin: TextIO, stdout: TextIO, argv: List[str]) -> None:
    words: List[str] = list(get_words(stdin))
    words_dict: Dict[str, int]
    keywords: List[str]
    words_dict, keywords = compile_dictionary(words)
    pointers: Generator[int, None, None] = compile_pointers(words, words_dict)
    write_dictionary(stdout, keywords)
    write_pointers(stdout, pointers)

if __name__ == "__main__":
    stdin: TextIO
    stdout: TextIO
    with get_std_streams(sys.argv) as (stdin, stdout):
        main(stdin, stdout, sys.argv)
```

## Testing

**#test_sorted_compression.py**

```
import unittest

from test_readable_compression import get_input_result, get_output_result

from sorted_compression import *

class TestSortedCompression(unittest.TestCase):
    def test_compile_dictionary(self) -> None:
        self.assertEqual(compile_dictionary(
                          ["FOO", "BAR", "FOO", "BAR", "FOO", "EGGS"]),
            ({"FOO": 0, "BAR": 1, "EGGS": 2}, ["FOO", "BAR", "EGGS"]))
        self.assertEqual(compile_dictionary(["FOO"]),
            ({"FOO": 0}, ["FOO"]))
        self.assertEqual(compile_dictionary([]), ({}, []))

    def test_compile_pointers(self) -> None:
        self.assertEqual(list(compile_pointers(["FOO", "BAR"],
            {"FOO": 1, "BAR": 0})), [1, 0])
        self.assertEqual(list(compile_pointers(["ONE", "TWO", "ONE"],
            {"ONE": 0, "TWO": 1})), [0, 1, 0])
        self.assertEqual(list(compile_pointers([], {})), [])
```

# Approach #3: bytes_compression.py (using the full range of a byte to store an integer)

Now so far, I have been encoding each integer using just the digits from 0-9, and a space between each integer.. However, each character of the output is a byte, and so could be any of 256 values (0-255). I can use these values to encode the integers instead. To use the same sort of idea, I can encode the integers in base 255 (using the digits 0-254) and use 255 as my ‘space’ delimiter. This approach will only need a new broad algorithm for doing the encoding of the integers, which can be done by write_pointers. I will also write a function to convert an integer into its new binary representation. As I am now working with the values of the actual bytes, not with their associated ASCII characters, it would make sense to write

to output in binary mode. I will not need to make any changes to the logic of compile_dictionary and compile_pointers.

## Algorithm #1: to_base (a general algorithm to convert an integer to a list of digits in its new base)

A short definition of a base (or radix) in a positional number system from Wikipedia:
"In the system with radix 13, for example, a string of digits such as 398 denotes the number $3 \times 13^2 + 9 \times 13^1 + 8 \times 13^0$.
More generally, in a system with radix b (b > 1), a string of digits $d_1 \ldots d_n$ denotes the number $d_1b^{n-1} + d_2b^{n-2} + \ldots + d_nb^0$, where $0 \leq d_i < b$".

So an integer can be represented as a series of integer digits in what is called a base.

This function will need to convert any positive integer

Because I am going to implement it as a generator, the function's output for demonstration will need to be wrapped in the list constructor.

Some example behaviour is:

```
>>> list(to_base(2, 10))
[0, 1, 0, 1]
>>> list(to_base(16, 256))
[0, 0, 1]
>>> list(to_base(1234, 10))
[10]
>>> list(to_base(2, 0))
[0]
```

This function doesn't really have edge cases where it is expected to function, as it should work on any positive integer, and the range of positive integers isn't very interesting..

To find the least significant digit (units digit), I can take the number modulo the base. The rest of the digits can be found by dividing the remaining number by the base, and repeating this process. I could implement this in some sort of while loop, but as the function repeats itself in this manner I can implement it recursively. An example of how this might work:

Convert the number 5 (101) to binary (base 2).
5 (101) modulo 2 is 1, this is the last digit.
Taking this digit away leaves 4 (100), dividing by 2 gives 2 (10).
2 (10) modulo 2 is 0, this is the second to last digit.
Taking this digit away leaves 2 (10), dividing by 2 gives 1 (1).
1 (1) modulo 2 is 1, this is the third to last digit.
Taking this digit away leaves 0, which divided by 2 is 0, so we know we are finished.

We have determined that 5 in binary must be 101

Note that for the implementation, I will yield what we normally consider the “last” digit - the units digit - first, as that is the order in which the function works. This is a sort of little-endian encoding, but with a non-binary base. It is also very simple to then decode a number from digits in this order, as you do not need to make any assertions about the number of digits - you know that the first digit will be units, and the next will be the base number, and the next the base number squared, and so on.

This algorithm works like this in pseudocode:

```
find the number modulo the base
this is the last digit
subtract the last digit from the number
divide the number by the base
if the new number is not 0:
    apply this process to the new number
```

It can be implemented like so in Python:

```python
def to_base(base: int, num: int) -> Generator[int, None, None]:
    quot: int
    rem: int
    quot, rem = divmod(num, base)
    yield rem
    if quot:
        yield from to_base(base, quot)
```

This uses Python’s divmod function, which does exactly what we’re after - finding both the quotient and the remainder. It is equivalent to finding the remainder modulo the base through the modulo function and then subtracting this from the number and dividing by the base to find the quotient, but a lot more concisely. It is implemented as a generator, even though it doesn’t need the lazy aspect of a generator. It is simply easier to implement like this, as the generator syntax is very expressive. Because it is implemented as a generator, I can’t just return or yield the value of the recursively called generator. This would result in, at the top level, a generator that first yields a digit and then yield a generator (which does the same, etc etc). The generator needs to be “unpacked”, so the whole function becomes one smooth generator. This can be done with Python’s yield from expression, which allows “generator delegation” - what I’m after.

## Algorithm #2: encode_pointer (encoding a pointer as some bytes)

This algorithm in this approach will be very short, but it will make sense to put this algorithm in a function like this in anticipation for future approaches.

It will need to find the values of the digits of the pointer in base 255 (with values from 0-254), and then use these to make a bytes object. A bytes object is like a string, but specifically for

bytes rather than characters. Bytes and bytearray are Python's two objects that do this. Bytearray, however, is mutable, which is not a function I need for this application. Making the bytes object after having found the digits in base 255 is trivial, as bytes accepts as a constructor argument "an iterable of integers in the range 0 <= x < 256, which are used as the initial contents of the array", which is exactly what I need.

Here is some example expected behaviour:

```
>>> list(encode_pointer(90))
[90]
>>> list(encode_pointer(255))
[0, 1]
>>> list(encode_pointer(256))
[1, 1]
>>> list(encode_pointer(0))
[0]
>>> list(encode_pointer(255 ** 2))
[0, 0, 1]
```

Even though this does so precious little, I will write it in pseudocode first:

```
find the digits of the pointer in base 255
use these digits to build a bytes object
```

This can then be done like so in Python:

```
def encode_pointer(pointer: int) -> bytes:
    return bytes(to_base(255, pointer))
```

## Algorithm #3: write_dictionary

Actually, I will be using exactly the same algorithm and approach for this function as previously in readable_compression.write_dictionary, so will not need to go over the algorithm and pseudocode of the algorithm again.. However, as I am now working with a binary file, I need to rewrite readable_compression.write_dictionary to work with a binary file.

For this to work I will need to change a couple of things - I will need to join the words with the bytes object b" ", I will need to use the bytes object b"\n" to separate the unique words from the pointers, and I will need to convert all the words read from input from strings into bytes objects. This last part can be done with a generator expression again, similarly to write_pointers in this approach - but applying the bytes constructor instead of encode_pointer.

I can use the bytes constructor with a string like so:

```
>>> bytes("abc", encoding="ascii")
b'abc'
```

Here I use the ascii encoding of the string. I will assume that my file to be compressed is an ascii file for this assessment. If it was encoded in utf8, I could use “utf8” as the encoding parameter.

This comes together to the following in Python:

```
def write_dictionary(bin_out: BinaryIO, words: List[str],
      separator: BinaryIO = b" ", end: BinaryIO = b"\n") -> None:
    i: str
    bin_out.write(separator.join(bytes(i, encoding="ascii") for i in words) + end)
```

## Main

This function will actually be nearly identical to sorted_compression.main. The only difference is that it expects a binary file to write to. I can access the binary version of sys.stdout through sys.stdout.buffer. It does something different - this is because it uses different functions with the same name. Even if it was identical, I couldn’t just do from sorted_compression import main, as this function needs to be defined in the namespace that contains the functions it makes calls to.

```
def main(stdin: TextIO, stdout: BinaryIO, argv: List[str]):
    words: List[str] = list(get_words(stdin))
    words_dict: Dict[str, int]
    keywords: List[str]
    words_dict, keywords = compile_dictionary(words)
    pointers: List[int] = compile_pointers(words, words_dict)
    write_dictionary(stdout, keywords)
    write_pointers(stdout, pointers, encode_pointer, b"\xff")
```

Calling this on its own won’t make much sense as it will be garble. I can call it and then call the decompressor function to test if it is working properly on a system level.

**#bytes_compression.py**

```
"""
A script to compress a file similarly to sorted_compression.py, but using the
full range of a byte to encode a pointer
"""

import sys

from readable_compression import get_std_streams, get_words, write_pointers
from sorted_compression import compile_pointers, compile_dictionary

from typing import *
from typing.io  import *

def to_base(base: int, num: int) -> Generator[int, None, None]:
    """
    Convert a number to a base, using a recursive algorithm to find the last
    digit. Returns digits in reverse order
```

```
    Example usage:
    >>> list(to_base(2, 10))
    [0, 1, 0, 1]
    >>> list(to_base(16, 256))
    [0, 0, 1]
    >>> list(to_base(1234, 10))
    [10]
    >>> list(to_base(2, 0))
    [0]

    Parameters:
    base - int - value of the base to convert to
    num - int - number to be converted

    Return:
    Generator[int, None, None] - a generator of integers, which are digits in
     reverse order
    """

    quot: int
    rem: int
    quot, rem = divmod(num, base)
    yield rem
    if quot:
        yield from to_base(base, quot)

def encode_pointer(pointer: int) -> bytes:
    """
    Encode a pointer value to bytes. Converts number to base 255 and constructs
    a bytes object.

    Example usage (converting back to list for visual clarity):
    >>> list(encode_pointer(90))
    [90]
    >>> list(encode_pointer(255))
    [0, 1]
    >>> list(encode_pointer(256))
    [1, 1]
    >>> list(encode_pointer(0))
    [0]
    >>> list(encode_pointer(255 ** 2))
    [0, 0, 1]

    Parameters:
    pointer - int - the pointer value to be converted

    Return:
    bytes - the pointer encoded in a bytes object, which can be written to a
     file
    """

    return bytes(to_base(255, pointer))

def write_dictionary(bin_out: BinaryIO, words: List[str],
      separator: BinaryIO = b" ", end: BinaryIO = b"\n") -> None:
    """
    Write string words to a binary file, followed by a newline

    Example usage:
    >>> get_output_result(write_dictionary, [["FOO", "BAR"]], binary=True)
    b'FOO BAR\n'
    >>> get_output_result(write_dictionary, [["FOO"]], binary=True)
    b'FOO\n'
    >>> get_output_result(write_dictionary, [[]], binary=True)
    b'\n'
```

```
    Parameters:
    bin_out - BinaryIO - binary file to write to
    words - List[str] - list of words to write
    separator - BinaryIO - separator value between words

    Return:
    None
    """

    i: str
    bin_out.write(separator.join(bytes(i, encoding="ascii") for i in words) + end)

def main(stdin: TextIO, stdout: BinaryIO, argv: List[str]):
    words: List[str] = list(get_words(stdin))
    words_dict: Dict[str, int]
    keywords: List[str]
    words_dict, keywords = compile_dictionary(words)
    pointers: List[int] = compile_pointers(words, words_dict)
    write_dictionary(stdout, keywords)
    write_pointers(stdout, pointers, encode_pointer, b"\xff")

if __name__ == "__main__":
    stdin: TextIO
    stdout: BinaryIO
    with get_std_streams(sys.argv, out_binary=True) as (stdin, stdout):
        main(stdin, stdout, sys.argv)
```

## Testing

### #test_bytes_compression.py

```
import unittest

from test_readable_compression import get_input_result, get_output_result

from bytes_compression import *

class TestBytesCompression(unittest.TestCase):
    def test_to_base(self) -> None:
        self.assertEqual(list(to_base(2, 10)), [0, 1, 0, 1])
        self.assertEqual(list(to_base(16, 256)), [0, 0, 1])
        self.assertEqual(list(to_base(1234, 10)), [10])
        self.assertEqual(list(to_base(2, 0)), [0])

    def test_encode_pointer(self) -> None:
        self.assertEqual(list(encode_pointer(90)), [90])
        self.assertEqual(list(encode_pointer(255)), [0, 1])
        self.assertEqual(list(encode_pointer(256)), [1, 1])
        self.assertEqual(list(encode_pointer(0)), [0])
        self.assertEqual(list(encode_pointer(255 ** 2)), [0, 0, 1])

    def test_write_dictionary(self) -> None:
        self.assertEqual(get_output_result(write_dictionary, [["FOO", "BAR"]],
                         binary=True), b"FOO BAR\n")
        self.assertEqual(get_output_result(write_dictionary, [["FOO"]],
                         binary=True), b"FOO\n")
        self.assertEqual(get_output_result(write_dictionary, [[]],
                         binary=True), b"\n")
```

# Approach #3b: bytes_decompression.py (decoding a file encoded by bytes_compression.py)

This will need to decompress a file compressed by bytes_compression.py. It will require reading the list of unique words which are encoded in a binary file, and reading and decoding the pointers. After this is has the unique words and the pointers, so it can use decompress from readable_decompression.py to put this back together to recreate the file. In order to decode the pointers, it will need to be able to convert number from base 255.

## Algorithm #1: from_base (a general algorithm to decode a series of digits from a base to an integer)

Because of the choice I made to write digits in backwards order, this algorithm will be very very simple. If the digits were in normal order, the algorithm would have to look ahead at the size of the entire list of digits. However, as it starts with the units digit, each digit is worth simply the base number raised to the power of its index - using 0-based indexing, as the units digit must equal 1, which is $base^0$. This means that I can use enumerate again, using the index as the digit's power. The expression for the value of any one digit will be digit * base ** power, for each (power, digit) in enumerate(digits). I need to find the total value of the digits, so I can use the sum function to calculate the sum of this generator.

```
def from_base(base: int, digits: Iterable[int]) -> int:
    digit: int
    power: int
    return sum(digit * base ** power for power, digit in enumerate(digits))
```

## Algorithm #2: decode_pointer (decode a pointer in a bytes object into an integer)

Again, as with bytes_compression.encode_pointer, this does very little. It doesn't even need to deconstruct the bytes object to a series of digits, because in Python when a bytes object is iterated over, the values of each byte are iterated over, so it can be treated as an Iterable[int], which is the argument needed by from_base.

```
def decode_pointer(pointer: bytes) -> int:
    return from_base(255, pointer)
```

## Algorithm #3: read_dictionary (read the list of unique words from the compressed file, and convert them to strings)

This algorithm is very very similar to readable_decompression.read_dictionary. Unfortunately, however, this needs to work with bytes objects, and while

readable_decompression.read_dictionary could already accept general arguments for the separator and end values, it needs to use an empty str or bytes literal to join the accumulated word. This is hardcoded into the function, and is probably not optimal design, but this is the way I've written it, so in the interest of time I will just write another function for working with bytes, rather than rewrite that function and everything already depending on it. Another thing it can do is to convert the bytes object to a str, by using the bytes.decode method.

```
def read_dictionary(in_file: BinaryIO, separator: BinaryIO = b" ",
      end: BinaryIO = b"\n") -> Generator[str, None, None]:
    c: bytes = in_file.read(1)
    word: List[bytes] = []
    while c != end:
        if c == separator:
            yield b"".join(word).decode("ascii")
            word = []
        else:
            word.append(c)
        c = in_file.read(1)
    if word:
        yield b"".join(word).decode("ascii")
```

## Algorithm #4: read_pointers (reading base 255 pointers from compressed file)

For the same reason as read_dictionary, this algorithm must be rewritten to work with bytes.

```
def read_pointers(in_file: BinaryIO) -> Generator[int, None, None]:
    c: bytes = in_file.read(1)
    n: List[bytes] = []
    while c:
        if c == b"\xff":
            yield decode_pointer(b"".join(n))
            n = []
        else:
            n.append(c)
        c = in_file.read(1)
    if n:
        yield decode_pointer(b"".join(n))
```

## Main

This main function again, quite simply needs to connect all the functions together appropriately.

```
def main(stdin: BinaryIO, stdout: TextIO, argv: List[str]) -> None:
    words: List[str] = list(read_dictionary(stdin))
    pointers: List[int] = read_pointers(stdin)
    decompress(stdout, pointers, words)
```

I can now call the compressor together with this.

```
$ echo "ASK NOt whAT your COUNTRY can do for you ask what you can do for your country" | python bytes_compression.py | python bytes_decompression.py
```

```
ASK NOT WHAT YOUR COUNTRY CAN DO FOR YOU ASK WHAT YOU CAN DO FOR YOUR COUNTRY
```

Although there was a slight problem here. This kind of syntax works in bash, but not in powershell where I was testing it. In powershell, this command lead to the following output:

```
Traceback (most recent call last):
  File "bytes_decompression.py", line 41, in <module>
    main(sys.stdin.buffer, sys.stdout)
  File "bytes_decompression.py", line 38, in main
    decompress(stdout, pointers, words)
  File "U:\Year 10\Computing\A453\task_2\readable_decompression.py", line 34, in decompress
    out_file.write(words[i])
IndexError: cannot fit 'int' into an index-sized integer
```

This actually meant that somehow it had read a number that was far too large from its compressed input. There was not actually any problem with my code, causing me to spend quite a bit of time fruitlessly trying to inspect my code and debug it. It turns out that Powershell is apparently very bad at feeding raw binary data through pipes. Eventually I inspected the data fed through the pipe to my decompressor, and found that some of the higher byte values had been turned into the byte value of a question mark. This gave me a specific problem to further investigate, and after some googling, I found on a number of sites, among which http://stackoverflow.com/questions/24708859/output-binary-data-on-powershell-pipeline, that Powershell tries to encode data using a certain encoding, and it does not support any encodings that won't corrupt binary data. This meant the separator bytes were being corrupted, so the compressed pointers had corrupted into one long pointer, and a number with a few digits in base 255 is bound to be very very large. This all means that in Powershell, this has to be slightly modified to

```
$ echo "ASK NOt whAT your COUNTRY can do for you ask what you can do for your country" | python bytes_compression.py --output compressed
$ python bytes_decompression.py --input compressed
```

with the same results. I will be using the syntax required in a Bourne-derived shell for future examples as I find it a lot cleaner and easier to follow, but be wary of this problem.

**#bytes_decompression.py**

```
"""
A script to decompress a file compressed by bytes_compression.py - with
```

```
pointers encoded using the full range of a byte.
"""

import sys

from readable_compression import get_std_streams
from readable_decompression import decompress

from typing import *
from typing.io import *

def from_base(base: int, digits: Iterable[int]) -> int:
    """
    Convert a series of digits into an integer from a given base, where the
    digits are in reverse order

    Example usage:
    >>> from_base(2, [0, 1, 0, 1])
    10
    >>> from_base(16, [15, 15])
    255
    >>> from_base(10, [1, 2, 3])
    321

    Parameters:
    base - int - the base to convert fromm
    digits - Iterable[int] - the series of number being converted

    Return:
    int - the value of the series of digits in the given base
    """

    digit: int
    power: int
    return sum(digit * base ** power for power, digit in enumerate(digits))

def decode_pointer(pointer: bytes) -> int:
    """
    Decode a pointer encoded in a bytes object into an integer value

    Example usage:
    >>> decode_pointer(b"\xfe")
    254
    >>> decode_pointer(b"\x00\x01")
    255
    >>> decode_pointer(b"\x0f")
    15

    Parameters:
    pointer - bytes - the pointer to be decoded

    Return:
    int - the decoded pointer value
    """

    return from_base(255, pointer)

def read_dictionary(in_file: BinaryIO, separator: BinaryIO = b" ",
      end: BinaryIO = b"\n") -> Generator[str, None, None]:
    """
    Reads and decodes into normal strings a list of unique words from binary
    file, until newline

    Example usage:
    >>> get_input_result(read_dictionary, b"ONE TWO THREE\nabc", [], wrapper=list,
binary=True)
```

```
    ['ONE', 'TWO', 'THREE']
    >>> get_input_result(read_dictionary, b"ONE\n", [], wrapper=list, binary=True)
    ['ONE']
    >>> get_input_result(read_dictionary, b"FOO BAR\n", [], wrapper=list,
binary=True)
    ['FOO', 'BAR']

    Parameters:
    in_file - BinaryIO - the file to read from

    Return:
    Generator[str, None, None] - generator of words read
    """

    c: bytes = in_file.read(1)
    word: List[bytes] = []
    while c != end:
        if c == separator:
            yield b"".join(word).decode("ascii")
            word = []
        else:
            word.append(c)
        c = in_file.read(1)
    if word:
        yield b"".join(word).decode("ascii")

def read_pointers(in_file: BinaryIO) -> Generator[int, None, None]:
    """
    Read and decode pointers from binary file until EOF

    Example usage:

    >>> get_input_result(read_pointers, b"\x03\xff\x02\xff\x01", [],
    ... wrapper=list, binary=True)
    [3, 2, 1]
    >>> get_input_result(read_pointers, b"\x02", [], wrapper=list, binary=True)
    [2]
    >>> get_input_result(read_pointers, b"\xfe", [], wrapper=list, binary=True)
    [254]

    Parameters:
    in_file - BinaryIO - file to read pointers from

    Return:
    Generator[int, None, None]
    """

    c: bytes = in_file.read(1)
    n: List[bytes] = []
    while c:
        if c == b"\xff":
            yield decode_pointer(b"".join(n))
            n = []
        else:
            n.append(c)
        c = in_file.read(1)
    if n:
        yield decode_pointer(b"".join(n))

def main(stdin: BinaryIO, stdout: TextIO, argv: List[str]) -> None:
    words: List[str] = list(read_dictionary(stdin))
    pointers: List[int] = read_pointers(stdin)
    decompress(stdout, pointers, words)

if __name__ == "__main__":
    stdin: BinaryIO
```

```
    stdout: TextIO

    with get_std_streams(sys.argv, in_binary=True) as (stdin, stdout):
        main(stdin, stdout, sys.argv)
```

## Testing

### #test_bytes_decompression.py

```
import unittest

from test_readable_compression import get_input_result, get_output_result

from bytes_decompression import *

class TestBytesDecompression(unittest.TestCase):
    def test_from_base(self) -> None:
       self.assertEqual(from_base(2, [0, 1, 0, 1]), 10)
        self.assertEqual(from_base(16, [15, 15]), 255)
        self.assertEqual(from_base(10, [1, 2, 3]), 321)

    def test_decode_pointer(self) -> None:
       self.assertEqual(decode_pointer(b"\xfe"), 254)
        self.assertEqual(decode_pointer(b"\x00\x01"), 255)
        self.assertEqual(decode_pointer(b"\x0f"), 15)

    def test_read_dictionary(self) -> None:
       self.assertEqual(get_input_result(read_dictionary, b"ONE TWO THREE\nabc",
                        [], wrapper=list, binary=True), ['ONE', 'TWO', 'THREE'])
        self.assertEqual(get_input_result(read_dictionary, b"ONE\n", [],
                            wrapper=list, binary=True), ['ONE'])
        self.assertEqual(get_input_result(read_dictionary, b"FOO BAR\n", [],
                            wrapper=list, binary=True), ['FOO', 'BAR'])

    def test_read_pointers(self) -> None:
       self.assertEqual(get_input_result(read_pointers,
                b"\x03\xff\x02\xff\x01", [], wrapper=list, binary=True), [3, 2,
1])
        self.assertEqual(get_input_result(read_pointers, b"\x02", [],
                wrapper=list, binary=True), [2])
        self.assertEqual(get_input_result(read_pointers, b"\xfe", [],
                wrapper=list, binary=True), [254])
```

For reasons elaborated on later, this is the last unit test I will write. This means I can test all of my code using

```
$ python -m unittest discover
```

```
.................
----------------------------------------------------------------------
----
Ran 16 tests in 0.002s

OK
```

This shows that my code is working.

# Approach #4: prefix_compression.py (using misaligned prefix codes to encode pointers)

This is the final improvement I will make for this task. To achieve really high compression on a text, I've realised I need to be able to encode common words in as few bits as possible, while possibly sacrificing more bits for less common word. This means I need an encoding that can encode words with different lengths. This is already being done by bytes_compression and readable_compression, but in two fundamentally less useful ways. They both write codes using individual bytes rather than bits, making the codes "coarser", in a sense. They also can do variable lengths, but they do so by way of a delimiter - a special pattern that separates each code. This also wastes valuable space in the output.

A better type of variable length encoding is using a prefix encoding. From Wikipedia: A prefix code is a type of code system (typically a variable-length code) distinguished by its possession of the "prefix property", which requires that there is no whole codeword in the system that is a prefix (initial segment) of any other codeword in the system.

In short, this means that with a prefix coding system you can always recognise what the next code word is as it could never be the start of another codeword.

An example of a prefix encoding system is UTF-8, one character encoding defined by Unicode. The way it works is roughly that the first couple of bits are used to signify the length of the following code word. I will roughly be following this approach - my plan is to have a fixed number of bits at the start of each code word, which will indicate its length.

Along this functionality, for this approach I will need to develop a way to write misaligned binary data.

## Class: BinaryWriter (an object that can be sent bits and write them)

What this part of my code will need to do is accumulate bits until there are enough to write a byte, and then do so. This means it needs to keep track of state. This can most easily be done by using a class.

The idea is that the class will have a file it knows to write to, and a 'buffer' of bits that it can add to, and then write from to the file. It should implement a few methods - one to write a single bit, which can then check if the buffer is full and if so, write a byte to the file and flush the buffer. There should also be a method to write multiple bits at once, so the program can be more intuitively called by writing a whole code, and the boilerplate of writing and flushing the buffer can be handled by the object.

It should be noted that this isn't the most optimum possible behaviour, and it represents a significant slowing down of my code, at this point, where calculation and logic has to be performed for each bit to write a file. However, any more optimal approaches would be pretty vastly complicated, and this problem is quite uncommon anyway - data is almost always stored using the byte architecture of a computer's memory system. Within this system, operations are usually very efficient, but adding a layer of complexity to that will slow down my code. This possibility will however allow me to massively improve the compression ratio.

The full class definition will be documented in the full source of my program, as always, but it consists of just these methods.

### Method #1: __init__ (the constructor)

In Python, the __init__ method is called when a new instance of an object is created. It is used like a constructor - it can 'set up' required structures and attributes. Because of the way objects in Python work, the first argument that will be passed to any method is the object instance which it is being called from. It is convention to name this argument 'self'. self can then be used to access and modify the given instance. Subsequent arguments given to the method will be the arguments after self. This object's __init__ needs to create the bit buffer it will be using, and have an argument which is a file-like object, which it can write to. It will assign this file to a property of the object, so it can be accessed by other methods.

Defining __init__ is actually overloading it - __init__ is a special name in Python which is used in certain circumstances. Similarly, I could overload an operator, like the + operator which is __add__. As you can see, these names all have double underscores.

```
def __init__(self, out_file: BinaryIO) -> None:
    self.bit_buffer: List[int] = []
    self.out_file: BinaryIO = out_file
```

### Method #2: _write_bit (taking a single bit and appending this to the bit buffer)

This method will be used to add a single bit to the bit buffer. This is implemented as a private method, as the intended use of this class is reading multiple bits. Privacy is indicated by starting the method name with an underscore. Note that there is no actual implementation of private methods in Python - this naming shows anyone using the class that the intended use is not for them to access that function, but they still can if they want to. This ties into the "we are all consenting adults here" philosophy, from earlier.

What this method does is take an integer argument, which is expected to be 0 or 1, and then append this to its buffer. If the length of the buffer is 8, it will call the _write_buffer method.

```
def _write_bit(self, bit: int) -> None:
```

```
        self.bit_buffer.append(bit)
        if len(self.bit_buffer) == 8:
            self._write_buffer()
```

### Method #3: _write_buffer (writing the bit buffer to file)

This will take the bit buffer, convert it from binary to an integer between 0 and 255 (as it is an 8 digit binary number), then convert this to a bytes object, and write it to the output file. Finally, it will need to empty the bit buffer. The first part of this can all be done in one line, using the later defined from_base, and then the bytes constructor. Note that there are square brackets around the from_base call, as the bytes constructor takes an iterable of integers from 0-255 (this makes sense as a bytes object is in itself an iterable of integers in this range, it just seems a little counter intuitive when only creating bytes objects of length 1). This is, again, implemented as a private method.

```
def _write_buffer(self) -> None:
    self.out_file.write(bytes([from_base(2, self.bit_buffer)]))
    self.bit_buffer[:] = []
```

### Method #4: write (writing an iterable of bits to the file)

This is now actually very simple, as the necessary checking is already implemented by write_bit. All this method needs to do is call write_bit for every given bit to write

```
def write(self, binary: Iterable[int]) -> None:
    for bit in binary:
        self._write_bit(bit)
```

### Method #5: flush (ensure remaining bits are written)

This method is important to remember - because of the way that a BinaryWriter buffers its output, if the program finishes while there are bits in the buffer, these will not have been written to the desired output file. This will result in desired information being lost, and will make decoding impossible. Therefore, this method to “flush” remaining bits is needed. It will work by writing 0s until the bit buffer has a length of 0, indicating it has been emptied. This can, again, be implemented using the truthiness of a list.

The user of this class will need to call this method when they are done, which isn’t entirely desirable behaviour but I will have to do it this way because of the whole structure that the BinaryWriter is in - as in, a user-defined class that is initialised and managed within the script.

```
def flush(self) -> None:
    while self.bit_buffer != 0:
        self._write_bit(0)
```

## Algorithm #1: padded_base (converting to a base with padding)

Because this approach will not be using delimited codes but prefix codes, I know that I will need to make conversions to bases using padding. In previous approaches, I have written down numbers as a human would, not using any leading zeroes. However, for this approach, I know that occasionally a number has to be padded like that. For example, in base 10, I might write the number 2 as 02. Happily, this algorithm is still very similar to my recursive bytes_compression.to_base. It can still be implemented recursively, but this time needs another argument - the number of expected digits. When I make a recursive call, I will have calculated one digit, so can subtract one from this number as I am expecting one less digit. This argument will now be my base case - when the required number of digits is 0, the function can stop recursing, as the caller does not expect any digits. This condition needs to be checked at the start of the function, rather than just before the recursive call.

Here is some sample behaviour:

```
>>> list(padded_base(10, 123, 5))
[3, 2, 1, 0, 0]
>>> list(padded_base(2, 10, 6))
[0, 1, 0, 1, 0, 0]
>>> list(padded_base(2, 7, 6))
[1, 1, 1, 0, 0, 0]
```

Again, there aren't really any edge cases to demonstrate.

Apart from the padded behaviour, the function stays much the same. In pseudocode, it will look something like this:

```
if more digits are required:

find the number modulo the base
this is the last digit
subtract the last digit from the number
divide the number by the base
decrease remaining expected number of digits
if more digits are required:
    apply this process to the new number
```

In Python:

```
def padded_base(base: int, num: int, pad: int) -> Generator[int, None, None]:
    if pad:
        quot, rem = divmod(num, base)
        yield rem
        yield from padded_base(base, quot, pad - 1)
```

## Algorithm #2: get_prefix_code (generate a prefix code from a pointer value)

Because of later considerations, I never actually use a function to generate a prefix code from a pointer value. However, for completeness of chronology and development process, it still makes sense to talk about.

I will outline my method for generating a prefix code from a pointer. First I will make some assumptions from other parts of my code. As my prefix codes will have different lengths, I assume that these preset lengths (boundaries) have been determined elsewhere, and will be passed to my function in a list of lengths. This is in fact all I need to assume.

From the length of the list of boundaries, I can extrapolate the number of bits that the codes use to indicate length - as I only need enough bits to distinguish between each index in the list of lengths, the number of bits will be the number of bits needed to store the largest index in the list. I can find the largest index from len(boundaries) - 1, and I can use the method bit_length of int to determine the number of bits needed to store it.

Given the number of bits that the codes use to indicate length, I can iterate upwards through the boundaries, and as soon as a boundary is large enough to contain my integer, I can use this boundary to generate my prefix code. I will return a padded binary length indicated chained to a padded binary value of the integer being encoded.

At this point, I will unfortunately not be providing examples of how this is expected to work. This is both because my functions are starting to get so complex that doing them by hand would take a lot of time, and I do not have much time of this controlled assessment left, so need to focus on coding and documenting. If I had more time, I would have provided examples.

```
def get_prefix_code(n: int, boundaries: List[int]): Iterable[int]:
    bits: int = (len(boundaries) - 1).bit_length()
    ind: int
    i: int
    for ind, i in enumerate(boundaries):
        if n.bit_length() <= i:
            return itertools.chain(binary(ind, bits), binary(n, i))
            break
```

However, there was a consideration that made get_prefix_code redundant. It might be simpler in code to call get_prefix_code for each pointer, but in reality it will be faster to precompute all of my prefix codes, for speed, as the intention is that there are repeating code words.

This has another few benefits: If I precompute in this sense, I can actually make use of more codes. Because of the way get_prefix code was written, some available prefix codes would never be used. I'll demonstrate this by simply modeling its process.

Say my chosen boundaries were [2, 4]. This means each code has one bit to indicate length, and then either 2 or 4 more bits.

Given the pointer value 0, get_prefix_code would see that this number is small enough to be encoded with the first boundary - 2 bits. It would therefore encode it as 000. We can see that the number 0 would never be encoded with the 4 bit boundary. So therefore, the code 10000 would never be used. This is because my encoding works with some actual calculation based on the code, in which 000 and 10000 would be held to be equivalent.

If I simply precompute, and generate codes in order and store a mapping of numbers to codes, I could say that 10000 comes after 011, so is number 4 (rather than 10100 being number 4). This allows for more codes to be used.

One more consideration is that there might be case where it is desirable to have multiple equal boundaries, which, using this system, can still produce different prefix codes.

I will implement the function as a generator, as this is easiest. The function's caller should then coerce this into a list, and can then use the list to encode integers used as indices in the list into pointers.

This function goes over each boundary, and systematically generates each code in the boundary's range. It uses itertools.chain to chain together the two generators of the prefix identifying the length of the codeword, and the codeword.

```
def generate_prefix_codes(boundaries: List[int]
    bits = (len(boundaries) - 1).bit_length()
    for jnd, j in enumerate(boundaries):
        for i in range(2 ** j):
            yield list(itertools.chain(padded_base(2, jnd, bits),
                       padded_base(2, i, j)))
```

## Algorithm #3: write_pointers (writing pointers and encoding with a prefix code mapping)

This needs to take a series of pointer values, and write these to a BinaryWriter. Because of all the other functions that I've written, this comes together to a pretty simple function. All it needs to do is take a list to use as a mapping for converting pointer values to pointers encoded as a prefix code, and then go over each pointer writing its encoded version to the BinaryWriter. In Python, it looks like this:

```
def write_pointers(out_binary: BinaryWriter, pointers: List[int],
      prefix_codes: List[List[int]]) -> None:
```

```
    for pointer in pointers:
        out_binary.write(prefix_codes[pointer])
```

## Algorithm #4: write_boundaries (writing the boundary value to file)

This function will need to take the boundaries and write these to the compressed file, as a kind of metadata. I could have had my decompressor also take boundaries from the user, but this would have felt a bit like cheating - this is information used in the encoding, so I will be storing it along with the compressed text. This is a pretty simple function, aside from that. All it needs to do is write a byte for each boundary, followed by a maximum value byte for the final boundary. This is because I will assume each boundary is lower than 255. This is a reasonable assumption to make, as with four boundaries of 254 I could encode 1157920892373161954235709850086879078532699846656405640394575840079131296 39936 pointers, which is more than enough for any reasonable text file. Making this assumption will greatly increase the simplicity of my code.

Because I want each boundary in one byte, and the boundaries are given in a list, I can actually use the bytes() constructor again to form a bytes object of boundaries, and concatenate this to a \xff byte, write this to the file and be done with it.

```
def write_boundaries(out_binary: BinaryIO, boundaries: List[int]) -> None:
    out_binary.write(bytes(boundaries) + b"\xff")
```

## Subroutine #1: get_boundaries (get the prefix boundaries from the user)

This function will need to interact with a user's input to obtain the prefix boundaries. Like get_std_streams, I will get them from the command line arguments. This is why it was a good idea to use parse_known_args and then reassign to the value of args, as this function can now continue to parse the arguments. Again, a lot of the work can be done by the argparse module.

As this function interacts with 'dirty' user input, it will need to deal with input validation, and what to do with invalid input. On a basic level, argparse already detects invalid input, in the sense that by the way I define the argument, with type=int, it knows that arguments must be integers. However, there is another level on which the arguments must be valid - the boundaries must be high enough to encode all the pointers. I know that each boundary size can encode 2 ** size pointers, as it constitutes a binary digit of the length of its size. Therefore, the potential number of boundaries is the sum of 2 ** i for each i in boundaries. I can take an argument of how many pointers need to be encoded.

Something else that needs to be accounted for is what happens if the user does not input any boundaries at all. I could make the boundaries a required argument, but I would prefer my program to have the functionality of automatically providing arguments. If the user does not supply any arguments, the item in the returned namespace will be the sentinel value of None. This is the only possible falsy value (as the list of boundaries would necessarily have

a length greater than 1, and even if the list of boundaries could have a length of 0, this would also be invalid). I can therefore test the truthiness of the boundaries given, and if it is false (ie if not boundaries) I can supply values. I can then test the potential sum of the values against the required size, and if this is not large enough, I can also force an override. If both of these requirements have been met, only one more thing is left to do - sort the boundaries, as some parts of my code require the boundaries to be in increasing order. This can be done in place with boundaries.sort().

My method for generating boundaries is using a predetermined set of calculations. It is likely to not be a bad method, but it could probably use some improvement, by trying to find a pattern in the optimal boundary patterns for some sample files. However, I don't have the time inside of this assessment to do that, so I just used some exponentially increasing boundary values, as this would provide some smaller and larger codes for the program. To find the largest boundary required, I used math.log with a base of 2 on the required size (ie 2 raised to what power will be sufficient to produce the size). I rounded this up, using the int() constructor, which rounds down, +1.

I then create a list of one eighth of this value, one quarter, one half, and then the value (using integer division). Because some of these value may end up being 0, I add 1 to each number. I know that this will increase so I don't need to sort it.

My argument construction for the boundaries takes an argument name from the function caller. This is because of a reason seen later in my code, in lossless_compression. It specifies the type of the arguments as integers, and then uses 'nargs="+"'. This tells the argument parser that this flag expects a following list of one or more arguments (not unlike the meaning of + in regex).

Here, to find the value of the arguments read in the namespace, I can't just directly access the property through a hardcoded syntactical name, as the name is a variable. Therefore, I use the __getattribute__ method to pass a string of the name of the attribute I want.

```
def get_boundaries(argv: List[str], size: int, name: str) -> List[int]:
    parser: argparse.ArgumentParser = argparse.ArgumentParser()
    parser.add_argument(f"--{name}", type=int, nargs="+")
    #parser.add_argument("--{}".format(name), type=int, nargs="+")
    args: argparse.Namespace
    remaining: List[str]
    args, remaining = parser.parse_known_args(argv)
    boundaries: List[int] = args.__getattribute__(name)
    argv[:] = remaining

    needs_override: bool = False

    if not boundaries:
        needs_override = True
    else:
        potential: int = sum(2 ** i for i in boundaries)
        if potential < size:
            needs_override = True
            sys.stderr.write("boundaries too small. entering defaults")
        else:
            boundaries.sort()
```

```
    if needs_override:
        largest_bound: int = int(math.log(size + 1, 2) + 1)
        boundaries = [largest_bound // 8 + 1,
                      largest_bound // 4 + 1,
                      largest_bound // 2 + 1,
                      largest_bound]

    return boundaries
```

## Main

The main function here, as normal, mainly just ties together functions appropriately. This main function, however, also does something extra. As I am binary misaligned, it would be easier for a receiving program to just read until it reaches a special EOF marker codeword. Therefore, I add an EOF to the end of my word dictionary, and when calling compile_pointers, append EOF to my words. EOF is a globally defined constant here. It has the value of -1, which is quite a commonly used sentinel value for EOF, and this is allowed in a list of strings in Python because Pythonic lists can be heterogenous.

At the end, I remember to call my BinaryWriter's flush method, to ensure all the desired bits to write are written.

```
def main(stdin: TextIO, stdout: BinaryIO, argv: List[int]) -> None:
    bw: BinaryWriter = BinaryWriter(stdout)
    words: List[str] = list(get_words(stdin))
    words_dict: Dict[str, int]
    keywords: List[str]
    words_dict, keywords = compile_dictionary(words)
    words_dict[EOF] = len(words_dict)
    prefix_boundaries: List[int] = get_boundaries(argv,
                                      len(keywords), "boundaries")
    prefix_codes: List[List[int]] = list(generate_prefix_codes(prefix_boundaries))
    pointers: Generator[int, None, None] = compile_pointers(words + [EOF],
                                                    words_dict)
    write_boundaries(stdout, prefix_boundaries)
    write_dictionary(stdout, keywords)
    write_pointers(bw, pointers, prefix_codes)
    bw.flush()
```

**#prefix_compression.py**

```
"""
A script to perform lossy compression on a text, encoding pointers to words
with a prefix encoding system.
"""

import sys
import itertools
import argparse
import math

from readable_compression import get_std_streams, get_words
from sorted_compression import compile_pointers, compile_dictionary
from bytes_compression import write_dictionary
from bytes_decompression import from_base

from typing import *
from typing.io import *
```

```
def get_boundaries(argv: List[str], size: int, name: str) -> List[int]:
    """
    A function to parse a set of boundaries to use for prefix encoding from
    given arguments. It also takes the size of the prefix encoding that is
    required, and will automatically generate boundaries if the user supplies
    insufficient boundaries or doesn't supply boundaries.

    Example usage:
    >>> get_boundaries(["--boundaries", "1", "2"], 3, "boundaries")
    [1, 2]
    >>> get_boundaries(["--boundaries", "4"], 3, "boundaries")
    [4]
    >>> get_boundaries(["--bounds", "82"], 400, "bounds")
    [82]
    >>> get_boundaries(["--bounds", "1", "2"], 400, "bounds")
    [2, 3, 5, 9]

    Parameters:
    args - List[str] - list of arguments to parse
    size - int - size of encoding required (largest value that needs
     to be encoded)
    name - str - name of the flag used in the arguments

    Return:
    List[int] - a list of boundaries, which are in numerical order

    """

    parser: argparse.ArgumentParser = argparse.ArgumentParser()
    parser.add_argument(f"--{name}", type=int, nargs="+")
    #parser.add_argument("--{}".format(name), type=int, nargs="+")
    args: argparse.Namespace
    remaining: List[str]
    args, remaining = parser.parse_known_args(argv)
    boundaries: List[int] = args.__getattribute__(name)
    argv[:] = remaining

    needs_override: bool = False

    if not boundaries:
        needs_override = True
    else:
        potential: int = sum(2 ** i for i in boundaries)
        if potential < size:
            needs_override = True
            sys.stderr.write("boundaries too small. entering defaults")
        else:
            boundaries.sort()

    if needs_override:
        largest_bound: int = int(math.log(size + 1, 2) + 1)
        boundaries = [largest_bound // 8 + 1,
                      largest_bound // 4 + 1,
                      largest_bound // 2 + 1,
                      largest_bound]

    return boundaries

EOF: int = -1

class BinaryWriter:
    """
    A class to write individual bits to a file, by handling a bit buffer, which
    is converted to a byte as soon as it reaches 8 bits
    """
```

```
    def __init__(self, out_file: BinaryIO) -> None:
        self.bit_buffer: List[int] = []
        self.out_file: BinaryIO = out_file

    def _write_bit(self, bit: int) -> None:
        self.bit_buffer.append(bit)
        if len(self.bit_buffer) == 8:
            self._write_buffer()

    def _write_buffer(self) -> None:
        self.out_file.write(bytes([from_base(2, self.bit_buffer)]))
        self.bit_buffer[:] = []

    def write(self, binary: Iterable[int]) -> None:
       for bit in binary:
            self._write_bit(bit)

    def flush(self) -> None:
       While self.bit_buffer:
            self._write_bit(0)

def padded_base(base: int, num: int, pad: int) -> Generator[int, None, None]:
    """
    Convert a number to a base, with padding

    Example usage:
    >>> list(padded_base(10, 123, 5))
    [3, 2, 1, 0, 0]
    >>> list(padded_base(2, 10, 6))
    [0, 1, 0, 1, 0, 0]
    >>> list(padded_base(2, 7, 6))
    [1, 1, 1, 0, 0, 0]


    Parameters:
    base - int - base to convert to
    num - int - number to convert
    pad - int - number of digits to produce

    Return:
    Generator[int, None, None] - generator of the digits in reverse order
    """



    if pad:
        quot, rem = divmod(num, base)
        yield rem
        yield from padded_base(base, quot, pad - 1)

#def get_prefix_code(n, boundaries):
#    bits = (len(boundaries) - 1).bit_length()
#
#    for ind, i in enumerate(boundaries):
#        if n.bit_length() <= i:
#            return itertools.chain(binary(ind, bits), binary(n, i))
#            break

def generate_prefix_codes(boundaries: List[int]
    """
    Generate all possible prefix codes from the boundaries in a deterministic
    order

    Parameters:
    boundaries - list of given prefix boundaries
```

```
    Return:
    Generator[List[int], None, None] - generator of all possible prefix codes
    """

    bits = (len(boundaries) - 1).bit_length()
    for jnd, j in enumerate(boundaries):
        for i in range(2 ** j):
            yield list(itertools.chain(padded_base(2, jnd, bits),
                       padded_base(2, i, j)))

def write_pointers(out_binary: BinaryWriter, pointers: List[int],
      prefix_codes: List[List[int]]) -> None:
    """
    encode a series of pointers using a given list of possible prefix codes, and
    write them to a BinaryWriter object.

    Parameters:
    out_binary - BinaryWriter - a BinaryWriter object to write to
    pointers - List[int] - a list of pointers to write

    Return:
    None
    """

    for pointer in pointers:
        out_binary.write(prefix_codes[pointer])

def write_boundaries(out_binary: BinaryIO, boundaries: List[int]) -> None:
    """
    Write some boundaries to a file, as a sort of metadata about the compression.
    It assumes that none of the boundaries are larger than 254, which for the
    purposes of compressing text is more than enough.

    Parameters:
    out_binary - BinaryIO - binary file to write to
    boundaries - List[int] - boundaries to write

    Return:
    None
    """

    out_binary.write(bytes(boundaries) + b"\xff")

def main(stdin: TextIO, stdout: BinaryIO, argv: List[int]) -> None:
    bw: BinaryWriter = BinaryWriter(stdout)
    words: List[str] = list(get_words(stdin))
    words_dict: Dict[str, int]
    keywords: List[str]
    words_dict, keywords = compile_dictionary(words)
    words_dict[EOF] = len(words_dict)
    prefix_boundaries: List[int] = get_boundaries(argv, len(keywords),
"boundaries")
    prefix_codes: List[List[int]] = list(generate_prefix_codes(prefix_boundaries))
    pointers: Generator[int, None, None] = compile_pointers(words + [EOF],
words_dict)
    write_boundaries(stdout, prefix_boundaries)
    write_dictionary(stdout, keywords)
    write_pointers(bw, pointers, prefix_codes)
    bw.flush()

if __name__ == "__main__":
    stdin: TextIO
    stdout: BinaryIO
    with get_std_streams(sys.argv, out_binary=True) as (stdin, stdout):
        main(stdin, stdout, sys.argv)
```

### Testing

As mentioned earlier, at this point, working through code by hand will take too long, so I am going to stop unit testing. However, I can still quite effectively test my code through some more system tests - if it performs well on a number of inputs, I can be confident in its correctness. There is little sense in testing much now though, as this program still only forms a part of the whole system (which is the compressor and decompressor)

## Approach #4b: prefix_decompression.py

This will need to decompress a file compressed by prefix_compression.py.

### Error type: EndOfBinaryFile (to call if the user of a BinaryReader reads past the end of the file)

For later, I will need to handle the exception of a user reading past the end of a binary file. To do this, I can create my own exception type, by subclassing Exception. This is all the class needs to do, as apart from that it can just behave as a normal exception. This makes this a very simple part of my code.

```
class EndOfBinaryFile(Exception):
    pass
```

### Class: BinaryReader (reading a binary file bit by bit)

#### Method #1: __init__ (constructor)

Constructs a BinaryReader, given the file to read from

```
def __init__(self, in_file: BinaryIO) -> None:
    self.bit_buffer: List[int] = []
    self.in_file: BinaryIO = in_file
```

#### Method #2: _read_byte_into_buffer (reads bytes into buffer)

Reads a byte from the input file. If the end of the file is reached, raises EndOfBinaryFile. Otherwise, converts the bit into binary digits and fills bit buffer.

```
def _read_byte_into_buffer(self) -> None:
    byte: bytes = self.in_file.read(1)

    if not byte:
```

```
            raise EndOfBinaryFile

        self.bit_buffer = list(padded_base(2, byte[0], 8))
```

### Method #3: read_bit (read a bit)

Check if the bit buffer is empty. If so, read a byte into the buffer. Otherwise, return the popped first element of the buffer.

```
def read_bit(self) -> None:
    if not self.bit_buffer:
        self._read_byte_into_buffer()
    return self.bit_buffer.pop(0)
```

### Method #4: read_bits (read a number of bits)

Quite simply, read a bit a number of times.

```
def read_bits(self, n: int) -> None:
    return [self.read_bit() for _ in range(n)]
```

## Algorithm #1: decompress (taking a list of integer pointer values and the list of unique words and using these to decompress the file)

This method needs to decompress a file, much like readable_decompression.decompress. However, this needs to take one more thing into account: it should stop at the EOF.

```
def decompress(out_file: TextIO, pointers: Iterable[int],
      words: List[str]) -> None:
    doSpace: bool = False
    i: int
    for i in pointers:
        dec: str = words[i]
        if dec == EOF:
            break

        if doSpace:
            out_file.write(" ")
        else:
            doSpace = True

        out_file.write(dec)
```

## Algorithm #2: read_prefix_code (reading a single prefix code from a file, given boundaries)

This function will need to read a prefix code from a BinaryReader. It is given the list of boundaries, so can again infer the number of bits being used to indicate length. It reads this many bits, finds the value of the length by converting the prefix read from base 2, and then

reads this many remaining bits. It then yields the entire prefix code including the prefix, and coerces it to a tuple. This is because the prefix code is desired as a tuple so that it can be used in a dictionary, as elaborated on in generate_translator.

```
def read_prefix_code(in_binary: BinaryReader, boundaries: List[int]) -> Tuple[int]:
    bits: int = (len(boundaries) - 1).bit_length()

    pref: List[int] = in_binary.read_bits(bits)
    pref_val: int = from_base(2, pref)

    return tuple(pref + in_binary.read_bits(boundaries[pref_val]))
```

## Algorithm #3: generate_translator (generate a translator dictionary from a list of boundaries

This will need to generate a dictionary with which to decode prefix codes. The work in generating prefix codes is already done by generate_prefix_codes, so it can just use a simple dictionary comprehension to make a dictionary mapping from prefix codes to their indices used by prefix_compression. Note that it must coerce each prefix code to a tuple. This is because tuple is Python's immutable equivalent of a list, and a dictionary requires the key value to be immutable. This is because mutable objects cannot be hashed in Python, which is so that the value of a key cannot be modified.

```
def generate_translator(boundaries: List[int]) -> Dict[Tuple[int], int]:
    i: List[int]
    ind: int
    return {tuple(i): ind for ind, i in
            enumerate(generate_prefix_codes(boundaries))}
```

## Algorithm #4: read_pointers (reading encoded pointers from file)

This will need to read pointers from a file. This is quite a simple task, as generate_translator and read_prefix code do all the work required for reading and decoding a prefix code. It expects the caller to have generated a translator dictionary with generate_translator, and expects this as an argument.

```
def read_pointers(in_binary: BinaryReader, boundaries: List[int],
     translator: Dict[Tuple[int], int]) -> Generator[int, None, None]:
   while True:
        pointer: int = translator[read_prefix_code(in_binary, boundaries)]
        yield pointer
```

## Algorithm #5: read_boundaries (read the meta prefix boundaries from the compressed file)

This function will need to read the prefix boundaries I decided to encode in prefix_compression.py. This is quite simply done by reading bytes until a \xff byte is encountered, and returning their value (which can be found with the ord function).

```
def read_boundaries(in_binary: BinaryIO) -> Generator[int, None, None]:
    c: bytes = in_binary.read(1)
    while c != b"\xff":
        yield ord(c)
        c = in_binary.read(1)
```

## Main

The main function, as always, ties together the appropriate functions. It must be sure to read things in the same order that they were written by prefix_compression (ie boundaries, then unique words, then pointers).

```
def main(stdin: BinaryIO, stdout: TextIO, argv: List[str]) -> None:
   br: BinaryReader = BinaryReader(stdin)
    prefix_boundaries: List[int] = list(read_boundaries(stdin))
    translator: Dict[Tuple[int], int] = generate_translator(prefix_boundaries)
    words: List[str] = list(read_dictionary(stdin)) + [EOF]
    pointers: List[int] = read_pointers(br, prefix_boundaries, translator)
    decompress(stdout, pointers, words)
```

**#prefix_decompression.py**

```
"""
A script to perform lossy decompression on text compressed by prefix_compression.py
"""

import sys

from readable_compression import get_std_streams
from prefix_compression import padded_base, from_base, generate_prefix_codes, EOF
from bytes_decompression import read_dictionary

from typing import *
from typing.io import *

class EndOfBinaryFile(Exception):
    """
    Exception for when BinaryReader reacher the end of a file
    """

    pass

class BinaryReader:
    """
    A class to read bits individually from a binary file by managing a bit buffer
    """

    def __init__(self, in_file: BinaryIO) -> None:
       self.bit_buffer: List[int] = []
        self.in_file: BinaryIO = in_file

    def _read_byte_into_buffer(self) -> None:
       byte: bytes = self.in_file.read(1)

        if not byte:
            raise EndOfBinaryFile

        self.bit_buffer = list(padded_base(2, byte[0], 8))

    def read_bit(self) -> None:
```

```
        if not self.bit_buffer:
            self._read_byte_into_buffer()

         return self.bit_buffer.pop(0)

    def read_bits(self, n: int) -> None:
        return [self.read_bit() for _ in range(n)]

def decompress(out_file: TextIO, pointers: Iterable[int],
      words: List[str]) -> None:
    """
    Decompress a series of pointer values, using a list of words, and write them
    to an output file.

    Parameters:
    out_file - TextIO - file to write to
    pointers - Iterable[int] - pointers to decompress
    words - List[str] - index of words to decompress them with

    Return:
    None
    """

    doSpace: bool = False
    i: int
    for i in pointers:
        dec: str = words[i]
        if dec == EOF:
            break

        if doSpace:
            out_file.write(" ")
        else:
            doSpace = True

        out_file.write(dec)

def read_prefix_code(in_binary: BinaryReader, boundaries: List[int]) -> Tuple[int]:
    """
    Read the next prefix code from file, by reading the prefix, and examining
    it to determine how far to read, and then returning the prefix along with
    the value.

    Parameters:
    in_binary - BinaryReader - BinaryReader to read bits from
    boundaries - List[int] - the list of what the boundaries are

    Return:
    Tuple[int] - this is so that this prefix code can be hashed, so its value
     can be retrieved from a dictionary
    """

    bits: int = (len(boundaries) - 1).bit_length()

    pref: List[int] = in_binary.read_bits(bits)
    pref_val: int = from_base(2, pref)

    return tuple(pref + in_binary.read_bits(boundaries[pref_val]))

def generate_translator(boundaries: List[int]) -> Dict[Tuple[int], int]:
    """
    Generate a dictionary from pointer tuples to their values.

    Parameters:
    boundaries - List[int] - the prefix boundaries to use
```

```
    Return:
    Dict[Tuple[int], int] - a dictionary mapping from tuples to integers, which
     are prefix codes to their values.
    """

    i: List[int]
    ind: int
    return {tuple(i): ind for ind, i in
            enumerate(generate_prefix_codes(boundaries))}

def read_pointers(in_binary: BinaryReader, boundaries: List[int],
     translator: Dict[Tuple[int], int]) -> Generator[int, None, None]:
    """
    Read pointers from a BinaryReader, indefinitely.

    Parameters:
    in_binary - BinaryReader - BinaryReader to read from
    boundaries - List[int] - list of boundaries to use
    translator - Dict[Tuple[int], int] - dictionary to translate prefix codes with

    Return:
    Generator[int, None, None] - generator of pointer values
    """

   while True:
        pointer: int = translator[read_prefix_code(in_binary, boundaries)]
        yield pointer

def read_boundaries(in_binary: BinaryIO) -> Generator[int, None, None]:
    """
    Read boundaries from binary file.

    Parameters:
    in_binary - BinaryIO - binary file to read from

    Return:
    Generator[int, None, None] - a generator of the boundaries found
    """

    c: bytes = in_binary.read(1)
    while c != b"\xff":
        yield ord(c)
        c = in_binary.read(1)

def main(stdin: BinaryIO, stdout: TextIO, argv: List[str]) -> None:
   br: BinaryReader = BinaryReader(stdin)
    prefix_boundaries: List[int] = list(read_boundaries(stdin))
    translator: Dict[Tuple[int], int] = generate_translator(prefix_boundaries)
    words: List[str] = list(read_dictionary(stdin)) + [EOF]
    pointers: List[int] = read_pointers(br, prefix_boundaries, translator)
    decompress(stdout, pointers, words)

if __name__ == '__main__':
    stdin: BinaryIO
    stdout: TextIO
    with get_std_streams(sys.argv, in_binary=True) as (stdin, stdout):
        main(stdin, stdout, sys.argv)
```

## Testing

```
echo "ASK NOt whAT your COUNTRY can do for you ask what you can do for your country" | python prefix_compression.py | python prefix_decompression.py
```

```
ASK NOT WHAT YOUR COUNTRY CAN DO FOR YOU ASK WHAT YOU CAN DO FOR YOUR COUNTRY
```

```
$ cat ../text/shakespeare.txt | head -500 | python prefix_compression.py | python prefix_decompression.py
```

```
...SECRET INFLUENCE COMMENT WHEN I PERCEIVE THAT MEN AS PLANTS INCREASE CHEERED AND CHECKED EVEN BY THE SELFSAME SKY VAUNT IN THEIR YOUTHFUL SAP AT HEIGHT DECREASE AND WEAR THEIR BRAVE STATE OUT OF MEMORY THEN THE CONCEIT OF THIS INCONSTANT STAY
```

(the dots indicate the output has been shortened)

```
$  cat ../text/shakespeare.txt | python prefix_compression.py | wc -c
```

```
1495045
```

```
$  cat ../text/shakespeare.txt | wc -c
```

```
5458199
```

This is a compression ratio of 0.28, which is not bad at all considering that I let the program use its default values.

# Task 3 - lossless compression and decompression

This task requires me to compress a text file and then “recreate the whole text”. In other words, I will need to perform lossless compression and decompression.

## Approach #1: lossless_compression.py (using the prefix encoding system to perform lossless compression)

This will be the only approach for the compression part of this task. This is because of how in depth I have done Task 2 - I have refined my system and approach, and I now can

actually modify prefix_compression.py to perform lossless compression with relatively little code, as much of it can be reused.

## Algorithm #1: get_runs (find runs of punctuation or words)

This function needs to identify the different 'runs' of punctuation and words, and yield these. It uses the globally defined constant sets LETTERS and PUNC to determine which category a character falls into. It finds the starting character's type, and then alternates as each run ends.

```
def get_runs(in_file: TextIO) -> Generator[str, None, None]:
    run: List[str] = []
    c: str = in_file.read(1)
    is_punc: bool = c in PUNC
    curr: Set[str] = PUNC if is_punc else LETTERS
    while c:
        if c in curr:
            run.append(c)
        else:
            yield "".join(run)
            run = [c]
            is_punc = not is_punc
            curr = PUNC if is_punc else LETTERS
        c = in_file.read(1)
    if run:
        yield "".join(run)
```

## Algorithm #2: separate_runs

As get_runs returns a muddled generator of runs, it will be useful to write a function to separate these into words and punctuation. It checks first if the list of runs has any length, so as not to encounter an IndexError when looking at the first item to check if it is punctuation. Based on the first index, it determines the starting index for punctuation and letter runs. It then uses a list slice to slice the list of runs. A slice is a more advanced form of indexing a list in Python, and it takes a step argument, for which I use 2, because the runs are alternating so each run of a type is two apart. It returns a tuple of a boolean signifying if the first run consists of punctuation, and each separated list of runs. If the list is empty, it arbitrarily returns True and two empty lists.

```
def separate_runs(runs: List[str]) -> Tuple[bool, List[str], List[str]]:
    if runs:
        start_punc: bool = runs[0][0] in PUNC
        if start_punc:
            punc_ind: int = 0
            letter_ind: int = 1
        else:
            punc_ind: int = 1
            letter_ind: int = 0

        return start_punc, runs[letter_ind::2], runs[punc_ind::2]
    else:
        return True, [], []
```

## Algorithm #3: write_pointers

The only writing function that needs changing from prefix_compression is write_pointers. This is because the pointers for each run type must be written together, so that each pointer can be decoded and printed as it is read. I can write them together, even if they use different boundaries, because the prefix property means that as long as I know whether I am expecting a word or punctuation run next, I can always read a code given its boundaries. Note that the starting run type therefore has to be encoded, which I can do with a single bit, as I am binary misaligned anyway, so this costs effectively nothing.

To write the runs in alternating fashion, I just use an if statement that uses sequence to determine the order to write runs in in each of its cases. As I am handling two iterators at once which may have a different length, I can't use a for loop. If they had the same length, I could zip() them. However, I will have to build my own iteration here using a while loop. Every time I get the next code using the next() function. As soon as a generator is exhausted and I call next() on it it will raise the StopIteration exception. If I put all of my calls to next() in a try statement, I can catch the StopIteration and use it to break from the while loop. This using of a try statement for control flow is not advised in many other languages, where the idea is to "Look Before You Leap". However, in Python, the philosophy is that it is "easier to ask for forgiveness than permission". In Python, try statements are optimised, and it is a good decision to use them for control flow. Note my code here means that StopIteration will only be raised when both generators are empty, because the generator which is being gone over first is always going to be of a length greater than or equal to the second one.

```
def write_pointers(out_binary: BinaryWriter,
      word_pointers: Generator[int, None, None], word_codes: List[List[int]],
      punc_pointers: Generator[int, None, None], punc_codes: List[List[int]],
      start_punc: bool) -> None:
    out_binary.write([1] if start_punc else [0])
    while word_pointers or punc_pointers:
        try:
            if start_punc:
                out_binary.write(punc_codes[next(punc_pointers)])
                out_binary.write(word_codes[next(word_pointers)])
            else:
                out_binary.write(word_codes[next(word_pointers)])
                out_binary.write(punc_codes[next(punc_pointers)])
        except StopIteration:
                break
```

## Main

The main function here becomes a little long winded. It needs to do most of the things prefix_compression.main does but for both types of run. I also add an EOF to both types of run, for simplicity.

```
def main(stdin: TextIO, stdout: BinaryIO, argv: List[str]) -> None:
    bw: binaryWriter = BinaryWriter(stdout)
    runs: List[str] = list(get_runs(stdin))
    start_punc: bool
```

```
    word: List[str]
    punc: List[str]
    start_punc, words, punc = separate_runs(runs)
    word_dict: Dict[str, int]
    keywords: List[str]
    words_dict, keywords = compile_dictionary(words)
    words_dict[EOF] = len(words_dict)
    punc_dict: Dict[str, int]
    keypunc: List[str]
    punc_dict, keypunc = compile_dictionary(punc)
    punc_dict[EOF] = len(punc_dict)
    word_boundaries: List[int] = get_boundaries(argv,
                                    len(keywords), "wboundaries")
    punc_boundaries: List[int] = get_boundaries(argv,
                                    len(keypunc), "pboundaries")
    word_codes: List[List[int]] = list(generate_prefix_codes(word_boundaries))
    punc_codes: List[List[int]] = list(generate_prefix_codes(punc_boundaries))
    word_pointers: List[int] = compile_pointers(words + [EOF], words_dict)
    punc_pointers: List[int] = compile_pointers(punc + [EOF], punc_dict)
    write_boundaries(stdout, word_boundaries)
    write_boundaries(stdout, punc_boundaries)
    write_dictionary(stdout, keywords)
    write_dictionary(stdout, keypunc, b"A", b"B")
    write_pointers(bw, word_pointers, word_codes,
                       punc_pointers, punc_codes, start_punc)
    bw.flush()
```

**#lossless_compression.py**

```
"""
A script to perform entirely lossless compresssion on an ascii text file, using
something similar to the prefix_code system from the lossy prefix_codes.py
"""

import sys
import string

from readable_compression import get_std_streams
from prefix_compression import (BinaryWriter, EOF, get_boundaries,
                                generate_prefix_codes, write_boundaries)
from sorted_compression import compile_dictionary, compile_pointers
from bytes_compression import write_dictionary

from typing import *
from typing.io import *

ALL_CHARS: Set[str] = {chr(i) for i in range(128)}
LETTERS: Set[str] = set(string.ascii_letters)
PUNC: Set[str] = ALL_CHARS - LETTERS

def get_runs(in_file: TextIO) -> Generator[str, None, None]:
    """
    Get runs of characters from an input file. These "runs" consist of either
    punctuation or letters, and necessarily alternate.

    Parameters:
    in_file - TextIO - file to read from

    Return:
    Generator[str, None, None] - a generator of runs
    """

    run: List[str] = []
    c: str = in_file.read(1)
    is_punc: bool = c in PUNC
```

```
    curr: Set[str] = PUNC if is_punc else LETTERS
    while c:
        if c in curr:
            run.append(c)
        else:
            yield "".join(run)
            run = [c]
            is_punc = not is_punc
            curr = PUNC if is_punc else LETTERS
        c = in_file.read(1)
    if run:
        yield "".join(run)

def separate_runs(runs: List[str]) -> Tuple[bool, List[str], List[str]]:
    """
    Separate a list of alternating runs into two lists, one of "words" and the
    other of punctuation. Also determines if the first run was punctuation or not.

    Parameters:
    runs - List[str] - the list of runs to be separated

    Return:
    Tuple[bool, List[str], List[str]] - a tuple of the two lists and the boolean
     indicating the type of the first run.
    """

    if runs:
        start_punc: bool = runs[0][0] in PUNC
        if start_punc:
            punc_ind: int = 0
            letter_ind: int = 1
        else:
            punc_ind: int = 1
            letter_ind: int = 0

        return start_punc, runs[letter_ind::2], runs[punc_ind::2]
    else:
        return True, [], []

def write_pointers(out_binary: BinaryWriter,
      word_pointers: Generator[int, None, None], word_codes: List[List[int]],
      punc_pointers: Generator[int, None, None], punc_codes: List[List[int]],
      start_punc: bool) -> None:
     """
    Write pointers to file. Writes pointers in alternating, fashion, as their
    respective runs were.

    Parameters:
    out_binary - BinaryWriter - BinaryWriter to write to
    word_pointers - Generator[int, None, None] - a generator of word pointers
     to be consumed
    word_codes - List[List[int]] - a list to encode word poiner values with
    punc_pointers - Generator[int, None, None] - a generator of puntuation
     pointers to be consumed
    punc - List[List[int]] - a list to encode puntuation poiner values with

    Return:
    None
    """

    out_binary.write([1] if start_punc else [0])
    while word_pointers or punc_pointers:
        try:
            if start_punc:
                out_binary.write(punc_codes[next(punc_pointers)])
                out_binary.write(word_codes[next(word_pointers)])
```

```
            else:
                out_binary.write(word_codes[next(word_pointers)])
                out_binary.write(punc_codes[next(punc_pointers)])
        except StopIteration:
                break

def main(stdin: TextIO, stdout: BinaryIO, argv: List[str]) -> None:
    bw: binaryWriter = BinaryWriter(stdout)
    runs: List[str] = list(get_runs(stdin))
    start_punc: bool
    word: List[str]
    punc: List[str]
    start_punc, words, punc = separate_runs(runs)
    word_dict: Dict[str, int]
    keywords: List[str]
    words_dict, keywords = compile_dictionary(words)
    words_dict[EOF] = len(words_dict)
    punc_dict: Dict[str, int]
    keypunc: List[str]
    punc_dict, keypunc = compile_dictionary(punc)
    punc_dict[EOF] = len(punc_dict)
    word_boundaries: List[int] = get_boundaries(argv,
                                    len(keywords), "wboundaries")
    punc_boundaries: List[int] = get_boundaries(argv,
                                    len(keypunc), "pboundaries")
    word_codes: List[List[int]] = list(generate_prefix_codes(word_boundaries))
    punc_codes: List[List[int]] = list(generate_prefix_codes(punc_boundaries))
    word_pointers: List[int] = compile_pointers(words + [EOF], words_dict)
    punc_pointers: List[int] = compile_pointers(punc + [EOF], punc_dict)
    write_boundaries(stdout, word_boundaries)
    write_boundaries(stdout, punc_boundaries)
    write_dictionary(stdout, keywords)
    write_dictionary(stdout, keypunc, b"A", b"B")
    write_pointers(bw, word_pointers, word_codes,
                       punc_pointers, punc_codes, start_punc)
    bw.flush()

if __name__ == "__main__":
    stdin: TextIO
    stdout: BinaryIO
    with get_std_streams(sys.argv, out_binary=True) as (stdin, stdout):
        main(stdin, stdout, sys.argv)
```

### Testing

Again, I can't really test this on its own, through unit testing or integrated system testing.

## Approach #1b: lossless_decompression.py (decompressing output from lossless_compression.py)

### Algorithm #1: read_decompress (read and decompress pointers)

Here, for two reasons, I am writing all of the new functionality in one function. This is because I am both running out of time, and it would start to be quite complicated to do this in

two functions, as it sort of needs to keep track of state as well. This function reads and decodes codes for both types of run, until it encounters an EOF, at which point it stops. It uses a similar approach to lossless_compression.write_pointers, in hardcoding the two different orders in which to read runs. It first reads bit signifying the type of a starting run which was written by lossless_compression.write_pointers to determine this.

```
def read_decompress(in_binary: BinaryReader, out_file: TextIO,
      word_boundaries: List[int], word_translator: Dict[str, int],
      words: List[str], punc_boundaries: List[int],
      punc_translator: Dict[str, int], punc: List[str]) -> None:
    start_punc: bool = in_binary.read_bits(1) == [1]
    while True:
        if start_punc:
            punc_code: List[int] = read_prefix_code(in_binary, punc_boundaries)
            punc_pointer: int = punc_translator[punc_code]
            punc_dec: str = punc[punc_pointer]
            if punc_dec == EOF:
                break
            out_file.write(punc_dec)

            word_code: List[int] = read_prefix_code(in_binary, word_boundaries)
            word_pointer: int = word_translator[word_code]
            word_dec: str = words[word_pointer]
            if word_dec == EOF:
                break
            out_file.write(word_dec)

        else:
            word_code: List[int] = read_prefix_code(in_binary, word_boundaries)
            word_pointer: int = word_translator[word_code]
            word_dec: str = words[word_pointer]
            if word_dec == EOF:
                break
            out_file.write(word_dec)

            punc_code: List[int] = read_prefix_code(in_binary, punc_boundaries)
            punc_pointer: int = punc_translator[punc_code]
            punc_dec: str = punc[punc_pointer]
            if punc_dec == EOF:
                break
            out_file.write(punc_dec)
```

## Main

This function ties together the other functions again. It is careful also to read all the data in the same order that it was written by lossless_compression.main.

```
def main(stdin: BinaryIO, stdout: TextIO, argv: List[str]) -> None:
    br: BinaryReader = BinaryReader(stdin)
    word_boundaries: List[int] = list(read_boundaries(stdin))
    word_translator: Dict[str, int] = generate_translator(word_boundaries)
    punc_boundaries: List[int] = list(read_boundaries(stdin))
    punc_translator: Dict[str, int] = generate_translator(punc_boundaries)
    words: List[str] = list(read_dictionary(stdin)) + [EOF]
    punc: List[str] = list(read_dictionary(stdin, b"A", b"B")) + [EOF]
    read_decompress(br, stdout, word_boundaries, word_translator, words,
                                punc_boundaries, punc_translator, punc)
```

**#lossless_decompression.py**

```
"""
A script to losslessly decompress a file encoded by lossless_compression.py
"""

import sys

from readable_compression import get_std_streams
from prefix_compression import EOF
from bytes_decompression import read_dictionary
from prefix_decompression import (BinaryReader, generate_translator,
                                  read_boundaries, read_prefix_code)

from typing import *
from typing.io import *

def read_decompress(in_binary: BinaryReader, out_file: TextIO,
      word_boundaries: List[int], word_translator: Dict[str, int],
      words: List[str], punc_boundaries: List[int],
      punc_translator: Dict[str, int], punc: List[str]) -> None:
    """
    Read and decompress pointers from file

    Parameters:
    in_binary - BinaryReader - BinaryReader to read from
    out_file - TextIO - text file to write to
    word_boundaries - List[int] - prefix boundaries for words
    word_translator - Dict[str, int] - dictionary to translate from words to
     pointer values
    words - List[str] - list of unique words from which ultimately the word is
     decoded
    punc_boundaries - List[int] - prefix boundaries for punctuation
    punc_translator - Dict[str, int] - dictionary to translate from punctuation to
     pointer values
    punc - List[str] - list of unique punctuation from which ultimately the punc is
     decoded

    Return:
    None
    """

    start_punc: bool = in_binary.read_bits(1) == [1]
    while True:
        if start_punc:
            punc_code: List[int] = read_prefix_code(in_binary, punc_boundaries)
            punc_pointer: int = punc_translator[punc_code]
            punc_dec: str = punc[punc_pointer]
            if punc_dec == EOF:
                break
            out_file.write(punc_dec)

            word_code: List[int] = read_prefix_code(in_binary, word_boundaries)
            word_pointer: int = word_translator[word_code]
            word_dec: str = words[word_pointer]
            if word_dec == EOF:
                break
            out_file.write(word_dec)

        else:
            word_code: List[int] = read_prefix_code(in_binary, word_boundaries)
            word_pointer: int = word_translator[word_code]
            word_dec: str = words[word_pointer]
            if word_dec == EOF:
                break
            out_file.write(word_dec)

            punc_code: List[int] = read_prefix_code(in_binary, punc_boundaries)
```

```
            punc_pointer: int = punc_translator[punc_code]
            punc_dec: str = punc[punc_pointer]
            if punc_dec == EOF:
                break
            out_file.write(punc_dec)

def main(stdin: BinaryIO, stdout: TextIO, argv: List[str]) -> None:
    br: BinaryReader = BinaryReader(stdin)
    word_boundaries: List[int] = list(read_boundaries(stdin))
    word_translator: Dict[str, int] = generate_translator(word_boundaries)
    punc_boundaries: List[int] = list(read_boundaries(stdin))
    punc_translator: Dict[str, int] = generate_translator(punc_boundaries)
    words: List[str] = list(read_dictionary(stdin)) + [EOF]
    punc: List[str] = list(read_dictionary(stdin, b"A", b"B")) + [EOF]
    read_decompress(br, stdout, word_boundaries, word_translator, words,
                                punc_boundaries, punc_translator, punc)

if __name__ == '__main__':
    stdin: BinaryIO
    stdout: TextIO
    with get_std_streams(sys.argv, in_binary=True) as (stdin, stdout):
        main(stdin, stdout, sys.argv)
```

## Testing

```
$ echo "ask NOT, wHat y0ur country CAN DO for you, ask what you can do
for your country" | python lossless_compression.py | python
lossless_decompression.py
```

```
ask NOT, wHat y0ur country CAN DO for you, ask what you can do for
your country
```

```
$ cat ../text/shakespeare.txt | head -500 | python
lossless_compression.py | python lossless_decompression.py
```

```
...
  Cheered and checked even by the self-same sky:
  Vaunt in their youthful sap, at height decrease,
  And wear their brave state out of memory.
  Then the conceit of this inconstant stay,
```

(the dots indicate that this output has been shortened).

```
$  cat ../text/shakespeare.txt | python lossless_compression.py | wc -c
2150980
$ cat ../text/shakespeare.txt | wc -c
5458199
```

This is actually a compression ratio of 0.39. This is not a bad result at all for my entirely lossless compression with a somewhat naive approach guided by the controlled assessment materials., and I am pleased with it.

# Development

I have documented much of my thought process in developing. I encountered one major problem, which was binary data with Powershell's pipes, which I have talked about. Apart from that, I didn't really encounter many bugs. Sometimes I had a SyntaxError or NameError, but these were generally from silly mistakes that I didn't feel were worth talking about. An example is when I misspelled "words_dict" while developing readable_compression.py. This lead to the following error:

```
Traceback (most recent call last):
  File "readable_compression.py", line 77, in <module>
    main(stdin, stdout)
  File "readable_compression.py", line 71, in main
    words_dict, pointers = compress_words(words)
  File "readable_compression.py", line 55, in compress_words
    pointers.append(words_dcit[word])
NameError: name 'words_dcit' is not defined
```

By reading the error, I could see exactly what the error was and at what line number it appeared, allowing me to fix this error very quickly. This kind of thing happened more often, but I decided not to continually document this.

# Afterthoughts

## Areas I would have liked to look at more, given more time:

I would have liked to look more at which compression techniques deliver which ratios, and look at how quickly each technique works.

Another thing I would have liked to investigate more was the optimal prefix boundaries. I used quite a naive approach to generate my boundaries, and I would have liked to run some analysis on a number of large files, to see if I could spot any patterns in optimal boundaries. I could have used a gradient search to search the space of boundaries for the one yielding the best compression ratio. I also know that written language is likely to follow "Zipf's law" so I think this would have had some potentially interesting results.

I would also have liked to have refined my code for the last task a bit, refactoring it and writing tests.

I also could not write as much documentation as I wanted to - at some point I ran out of time for writing pseudocode.

I would also have liked to look at more methods for compression. I was limited here by both the general form of the controlled assessment and time constraints, but other methods like LZW compression and Huffman encodings would have been interesting to look at, and compare to my method for prefix encoding.

## Sources

Throughout, I have made a lot of reference to Python's standard library documentation, for things like how to apply argparse. These can be found at https://docs.python.org/3/library/
At points I have also used http://stackoverflow.com/ for inspiration in some of my algorithms.
I have made reference to https://www.wikipedia.org/ for some of the more heavy theoretical mathematical and algorithmic material.

## Fun fact:

```
$  cat */*.py | wc -l
1686
```

Signifying I wrote 1686 lines of code for this controlled assessment.