CS 115 Homework 2: Decoding Poor Morse Code

In the mid-1800s, the electronic telegraph was invented. Capable of transmitting a signal over a large distance, it rapidly revolutionized communication worldwide.

However, the standard electronic telegraph had a major limitation: It can only send a single on-off signal. Thus, if we want to send more complicated messages like letters or words, we need ways of encoding them using only whether the channel is on or not.

One standard that emerged was the Morse Code: it encoded letters using a combination of short and long tones. Every letter got a unique sequence of these "dots" and "dashes", so by paying attention to how long the signal was on for, you could decode the letters. You would then write a word by encoding each letter (leaving a short pause between each dot and dash), and then having longer pauses between the individual encodings of the letters.

More specifically, the Morse code encodes each letter into the following sequences of dots (●) and dashes (▬).

A	● ▬	J	● ▬ ▬ ▬	S	● ● ●
B	▬ ● ● ●	K	▬ ● ▬	T	▬
C	▬ ● ▬ ●	L	● ▬ ● ●	U	● ● ▬
D	▬ ● ●	M	▬ ▬	V	● ● ● ▬
E	●	N	▬ ●	W	● ▬ ▬
F	● ● ▬ ●	O	▬ ▬ ▬	X	▬ ● ● ▬
G	▬ ▬ ●	P	● ▬ ▬ ●	Y	▬ ● ▬ ▬
H	● ● ● ●	Q	▬ ▬ ● ▬	Z	▬ ▬ ● ●
I	● ●	R	● ▬ ●		

This mechanism allows you to encode words purely as on-off signals, conveying information based only on how long and short the on and off signals are.

In this homework, you will implement several tasks related to the translation to and from Morse, with a focus on developing your skills at using recursion and higher-order functions in tandem. It will also test your ability to write unit tests to ensure your code is correct.

Remember: homework is designed to be completed over an extended period of time. Please log your process in `hw2_report.txt` More information is available under the labor log heading at the bottom of the file.

Task 1: Encoding into Morse

Your first challenge is to implement an encoding function, which converts a plain English word into Morse code.

You will be given a word, in the form of a string. Your goal will be to convert that word into Morse code, as a new string.

For this homework, we will be assuming a standard rule for conversion: Each letter is converted into the appropriate set of dashes and dots, represented using hyphens (-) and periods (.). Then, the encodings of the letters should be separated by a single space.

For example, the encoding of `"TEST"` is `"- . ... -"`.

Goal: Write the function `encode`. This takes in a string `plaintext`, which will be a single English word written in capital letters. The function must then return the encoding of that word in Morse code.

Some example input-output pairs are provided below.

```
>>> encode("IS")
".. ..."
```

```
>>> encode("GLOW")
"--. .-.. --- .--"
>>> encode("HISSES")
".... .. ... ... . ..."
```

For this function, you may assume that your input will always be a single word, no spaces, all capital letters. Your function does not have to handle invalid inputs.

Warning: For this problem set, you may not use loops! We want you to solve these problems using only higher-order functions and recursion. **If you use loops in your implementations, you will receive a 0 on this assignment!**

Task 2: Decoding Morse code

We now have the ability to encode into Morse code, but we also want to be able to do the reverse: Go from a Morse code encoding back to the English word it represents.

In other words, if we are given a string of dots and dashes, separated by spaces, we want to convert every consecutive section of dots and dashes into a single letter, using the same conversion table as before.

For example, given the string `"-.. . -- ---"`, we want to retrive the raw plaintext `"DEMO"`.

But what if the Morse code is invalid? For example, the Morse string `"..-..-"` does not map to any letter. Any such invalid string should be mapped to `"?"`, which we will use to represent any sequence we cannot accurately decode.

Task 2.1: Tests for Decoding

Before we start writing a decoder, we would like to have tests to ensure that our code is correct once it is finished.

Goal: In the file `test_hw2.py`, implement tests for the `decode` function. It should match the specifications above, taking in one string and returning the decoding.

Your tests should follow the naming scheme of `test_decode_<details>`, where the substring of `<details>` should cover what the test is checking for. For example, `test_decode_example` might check that the string for "EXAMPLE" is decoded properly.

Feel free to write as many tests as you feel necessary. Tests will be graded for completeness and correctness. Remember to cover all possible scenarios, and to account for edge cases.

Task 2.2: Decoding

Now that you have the tests for `decode`, you can start writing it, using your earlier tests to ensure that it is behaving correctly.

Goal: Write the function `decode`. This takes in a string `cyphertext`, which will be a valid Morse code encoding, as specified in the previous part. The function must then return the word corresponding to that encoding.

An example input-output pair is provided below.

```
>>> decode(". -..- .- -- .--. .-.. .")
"EXAMPLE"
```

Task 3: Decoding noisy Morse code

When you have nicely formatted Morse code, decoding it is a rather direct process. But in real life, communication is not that clean.

Maybe the operator is hasty, and doesn't leave enough time between letters, so their encodings flow together. Maybe their finger presses the button too long, so a dot becomes a dash. Maybe the signal goes down for a brief moment, splitting a dash into two dots. Or maybe one of the symbols is just plain dropped.

Whatever the reason, sometimes the encoding is not properly received. And that then presents a new challenge: Can we determine what the intended message was, despite the errors in transmission?

This is fairly complex, so for this problem, we will instead focus on a simpler problem.

Suppose that we have a valid encoding, but all the spaces have been deleted. For example, the string `".. ..."` (representing `"IS"`), would get corrupted to `"....."`. From this, you can recover `"IS"` by reinserting the space... but you can also reinsert the space incorrectly to get `".... ."`, and thus decode this as `"HE"`.

As `"HE"` and `"IS"` end up looking indistinguishable, we don't know what `"....."` is supposed to be. As such, you don't have to determine which one it is. Instead, you have to return a tuple with all the possible words that might have given this corrupted message.

As English is a large language, however, we have to limit our search. We will provide you with the tuple `dictionary`. This will contain a list of English words. When you are looking for words that a string might represent, you only have to check the words in this dictionary.

Goal: Write the function `matches`. This takes in a string `cyphertext`, which will be a Morse code encoding of some word, but with all the spaces removed. The function must then return a tuple of all possible words which could have resulted in that encoding.

Some example input-output pairs, using the default dictionary, are provided below.

```
>>> matches("---.-.")
("OR")
>>> matches(".....")
("HE", "IS", "SEE")
>>> matches(".-.-.-.-.-.-.-")
()
```

As the above example also shows, if there are no valid words, you should return an empty tuple.

Task 4: Analysing Collisions

After you are done, you might want to test with more words. Rather than editing the dictionary, we instead have provided you with two alternative dictionary files. By uncommenting the `from dict import dictionary` or `from bigdict import dictionary` lines, you will import a much larger dictionary, allowing for larger examples.

Note that to do this, **you must have the `dict.py` and `bigdict.py` files in the same directory/folder as your solution file**. So you should download those if you wish to use it.

However, with much larger dictionaries, we also can encounter more collisions. Before, we have seen that `"HE"` and `"IS"` are indistinguishable. Thus, we want you to analyse the dictionary a bit more, looking for other collisions.

Goal: Answer the following questions in the provided `hw2_report.txt`.

- For the following set of words, what "collisions" do they have? In other words, what words map to the same set of dots and dashes, such that they are indistinguishable once the spaces are removed?
  - "HIVE"
  - "MOOSE"
  - "SIRE"
  - "FENCE"
  - "ALSO"
- Find at least one other interesting pair of words forming a collision.
- Find at least one word that decodes uniquely, even if the spaces are removed.

Technical Comments: Python has a limit to recursion depth, and depending on how you set up your program, you might run into that limit. To fix this, add the following two lines to the top of your file:

```
import sys
sys.setrecursionlimit(10000) # Allows up to 10000 recursive calls
```

(Heads-up: On some copies of IDLE, `bigdict` is too large and crashes the program. If you find that your program crashes upon importing it, you might have to recomment that line and not use it.)

Submissions

Submit all files to the Gradescope assignment for Homework 2. You can access Gradescope through Canvas. You must submit the following files, with exactly the following file names:

- `hw2.py`
- `test_hw2.py`
- `hw2_report.txt`

Rubric

Category	Description	Points
Name	For stating your name on every file for submission.	2
Pledge	For writing the full Stevens Honor Pledge on every file for submission.	3
Labor log	For documenting your process of working on this homework.	5
Docstrings	For properly written and well-formatted docstrings.	10
Task 1: `encode`	For properly implementing `encode`.	10
Task 2.1: Tests	For having correct and comprehensive tests for the `decode` function.	20
Task 2.2: `decode`	For properly implementing `decode`.	20
Task 3: `matches`	For properly implementing `matches`.	20
Task 4: Report	For answering the required questions in the `hw2_report.txt`.	10
Total		100

Hints

- You might find writing a lookup function helpful. This should take in your dictionary tuple `morse`, takes in a letter, and finds the encoding of just that one letter.
- You might also find writing a reverse lookup function as well, which goes the other way around (starting at the encoding of one letter, and returning that letter).
- You may wish to review Python's string functions, which can be found here:

  https://docs.python.org/3/library/stdtypes.html#textseq.

  These can be quite helpful for this problem set, and can help you avoid "reinventing the wheel".
- For each task, you may find reusing your work from earlier tasks useful. You can use functions you have already written to avoid wasting time.
- You can use the keyword `in` to check membership in a tuple. `x in tup` will return `True` if and only if `x` appears in the tuple `tup`.
- While you have to use recursion for this problem set, higher-order functions can be quite handy, as they are very fast and powerful. In general, if you are processing hundreds or thousands of values, a higher-order function might be more appropriate than using recursion.

Suggestions and Reminders

- **Remember to include docstrings!** If your submission does not have docstrings, or has docstrings that are not related to your code, you will not get credit for them.

- Additionally, if your functions are more complicated than usual, you should add comments inside them explaining how they work. If you worry that you wouldn't know what your code is doing if you didn't write it, you should make sure that we can understand what it's doing by explaining it.
- **Plan out your code before you start writing it.** One of the objectives of this problem is to have you think about designing a more complicated program that involves several functions. You have complete autonomy in deciding what functions to write in order to implement the main tasks. Think carefully about which functions you will need and try to implement those functions so that they are as simple and clean as possible. Don't settle for the first thing you think of, and don't just jump right into writing code without thinking about how you'll organize your code. Strive to make your program simple and well organized.
- Our sample solution has around 7 functions, most of which around around 4 lines long. You are not required to have the same number of functions and you may have a few slightly longer functions; the point is that it's possible to complete this assignment without writing a lot of code. Don't get get scared by how much you have to write, and break down the problem into small, manageable chunks.
- **Until we tell you otherwise, you are not allowed to use loops!** We want you to learn to use recursion, so you are not allowed to use loops to solve these problems.
- **Don't forget to add your name and pledge!**