Project 2

EN.500.133 Bootcamp: Python

January 5, 2024

1 The Problem

Your stellar work at the Intelligence Agency has earned you a promotion to Spaceforce. You are part of the team that is sent to spy on the planet Namek. This planet is inhabited by humans who left Earth a few decades ago. Namekians are technologically advanced and do not use magnetic storage devices anymore. Instead of encoding their information as 0s and 1s, they store them in DNA. We'll use the typical notation here to represent the four nucleotide bases: 'A' to represent Adenine, 'C' for Cytosine, 'G' for Guanine, and 'T' for Thymine. Here is how they encode their information as a DNA sequence:

- Every character used in English text (lowercase, uppercase and punctuation) is represent by a number between 0 and 255 using ASCII Tables (see `http://www.asciitable.com/`)
- This number in binary form can be represented with 8 bits.
- As an example, "a" is encoded by the decimal number 97. It's binary representation would 01100001. These 8 bits can be encoded into 4 DNA bases following the scheme below. Following this scheme the letter "a" would be encoded in DNA bases as TCAT.

Binary pair	DNA base
00	A
01	T
10	C
11	G

- Your team has managed to steal a few DNA samples and a DNA sequencer that reads out the base pairs in order. Your task is to write a **Python *module*** named `dna_info.py` with functions that can decode the information stored in these DNA samples.

1.1 Encoder

Task 1: Write a function called encode_sequence which takes a string as an input. This function returns the sequence of DNA bases that contain the same information as the string. For instance, `encode_sequence("Frieza")` should return `TATCTGACTCCTTCTTTGCCTCAT`.

Hint : Python's in-built `ord()` and `chr()` functions provide a unique mapping between a character and an integer less than 255. `ord('a')` returns the integer 97 and `chr(97)` returns the character 'a'.

1.2 Decoder

Task 2: Write a function called decode_sequence which inverts DNA sequence back into English text. For example, `decode_sequence("TATCTGACTCCTTCTTTGCCTCAT")` should return `"Frieza"`.

1.3 Encryption

Before bringing the technology back to Earth, you have decided to add an extra layer of security. You plan to encrypt the DNA sequence using "exclusive-or" (also known as XOR) encryption before synthesizing and storing the molecules. The definition of the binary XOR operation, denoted with $\wedge$, is shown in Figure 1. Note that the XOR operation is symmetric, meaning that ordering of the operands is insignificant. In other words, for operands a and b, the value $a \wedge b = b \wedge a$.

Two inputs are required for encryption: (i) the string to encrypt and (ii) a key string. The steps you'll perform to encrypt an input string given a DNA-sequence-based key value are described as follows:

a	b	$a \wedge b$
A	A	A
A	T	T
A	C	C
A	G	G
T	T	A
T	C	G
T	G	C
C	C	A
C	G	T
G	G	A

Figure 1: The XOR function.

- Let k_1 be the first letter from the key and apply the XOR ($\wedge$) operation to k and ell_i for each letter ell_i in the input string. Concatenate the resulting letters together. For example, let the input string be TAAT, and the key be the string CAT. The first letter k_1 from the key is C. The output of this step would be $(T \wedge C)$ $(A \wedge C)$ $(A \wedge C)$ $(T \wedge C)$ = GCCG.

- Now, repeat the previous step, applying the next letter in the key, k_2, to the *output* from the previous step. In our example, this would mean applying the key letter C to the previous output GCCG.

- Repeat the this process until all letters in the key have been applied. The string resulting from the last round of XOR-ing is the encrypted message. In the example above with input string TAAT, and key CAT, the final encrypted message is CGGC.

Task 3: Write a function named `encrypt_decrypt` that takes as arguments a required string to be encoded and an optional string representing the key. The default value of the key should be set to CAT. The function should return the encrypted sequence.

A convenient feature of this XOR-based encryption is that, to decrypt the message, we can simply call the `encrypt_decrypt` function again, this time, passing in the encrypted sequence and the same key we used for encryption. As such, calling the function with an input string CGGC and using the key CAT, the output from the function will be TAAT, the string that we originally encrypted.

1.4 Error assessment

You are almost ready to market your technology back on Earth. (Never mind that this encryption scheme is not really very secure!) The last step is to ensure that a robot can take your encrypted sequence and synthesize the corresponding DNA molecule. The robot takes as input a string of letters (DNA bases). It starts from the first letter and synthesizes the corresponding base. Subsequent bases are attached to this chain via chemical reactions. Unfortunately, the robot's synthesis capability is not perfect. The table in Figure 2 describes the error associated with synthesizing a certain base.

	Actual Output Probability			
Desired Base	A	T	C	G
A	1.0	0.00	0.00	0.00
T	0.05	0.90	0.03	0.02
C	0.01	0.01	0.97	0.01
G	0.01	0.02	0.02	0.95

Figure 2: Synthesis output probabilities.

As noted, the robot is always successful adding Adenine (base 'A'). That is, if the robot receives input 'A', we are guaranteed that Adenine is synthesized and added to the chain with 100% accuracy. However, if 'T' is the input, the robot successfully synthesizes Thymine only 90% of the time; in other words, he robot creates the desired 'T' base correctly 90% of the time. Otherwise, it will synthesize Adenine (5% of the time) or Cytosine (3% of the time) or Guanine (2% of the time). The robot is 97% and 95% accurate synthesizing Cytosine and Guanine, respectively; the table supplies the remaining details.

To begin exploring how to address this situation, you'll need to write two methods:

- **Task 4: Write a function `synthesizer` that simulates this manufacturing process.** Your function should takes a sequence (string) as input and return the sequence (string) of DNA synthesized by the robot. You can use Python `random` module to generate random numbers (see https://docs.python.

org/3/library/random.html). `random.random()` generates a float in the range $[0, 1)$ uniformly at random.

- **Task 5: Write a function `error_count` that takes two strings as input, compares them letter by letter, and returns an integer that gives the number of mismatches found.**

1.5 Redundancy

You plan to solve the robot's inaccuracy problem with redundancy. For every encryption, you plan to repeat the synthesis n times. To read the information stored in DNA one has to read all n copies and compare them letter by letter (that is, base by base). If there were no errors in the manufacturing process, i.e., in the above table the diagonal elements are 1.0 and the rest of the elements are zero, then all n copies would be identical.

However since errors are present in your manufacturing process, a letter-by-letter comparison across all n copies may reveal that the copies are not identical. When this occurs, your strategy is to count the number of times a letter occurs in a specific position across all n copies and assume that the letter (base) that occurs most frequently is the "correct" one. To deploy this plan, you must write the following function.

- **Task 6: Write a function `redundancy` that takes as input an integer n, and an input string to synthesize, in that order. The function obtains n copies of the synthesized DNA from your `synthesizer` function. It then compares all n copies, to find the correct letter in each position, and returns the error-corrected string.**

In addition, to confirm that this strategy is useful, perform a little test and submit your results as described below:

- **Task 7: Use your `error_count` function to test if this scheme works as intended.** Pick a reasonably large sample input string (we recommend a minimum string length of 50 letters; more is generally better), and call your `redundancy` method on it. Tabulate how the error count goes down as n increases. (That is, try using your redundancy function on the same input string with different values of n and report the error count for each run.) Submit your evidence (a log of your output) as a PDF file (`error_count.pdf`) along with your code.

2 Instructions for submission

- As in project 1, when you submit your work, Gradescope will run your submitted code using a small number of tests to help you confirm that your solution is on the right track. Some test cases may be hidden until after the project deadline and so looking only at the auto-tested results from Gradescope is not a substitute for performing your own thorough testing.

- Submit two files to Gradescope for this assignment: (i) `dna_info.py` containing all six required methods and (ii) `error_count.pdf`. Be sure your name and JHED appear in a comment line at the top of `dna_info.py` and that they appear in your `error_count.pdf` file as well.

- Include substantial comments in your code. Your code will be evaluated not only on correctness, but also on style and good programming techniques. For instance, you should use loops and helper functions where appropriate. **Write docstrings for all functions.**

- If your solution is not working perfectly, turn it in as-is with detailed comments describing which parts are complete, and which still need work. Be sure that what you turn in runs, to make it possible to receive feedback and partial credit.

- You are strongly encouraged to adopt an incremental coding style, making small changes to your code one at a time, so that you always have a version of your program which runs, meaning you will always have something to turn in, even if it is not 100% complete. You are also reminded to exercise discipline in backing up your work. If you have trouble or need extra help, don't hesitate to attend office hours or contact a member of the course staff through Piazza.

- You are permitted to discuss the instructions with other students for clarification. However, you may not discuss any substance of your solution with anyone except course staff, whether that be algorithms, test cases, component design, or actual code.