In this lab we will gain experience in test-driven development by implementing a common bioinformatics function and its corresponding unittests by following a test-driven workflow.

**Step 1:**

Read the description and specification of the bioinformatics function on https://rosalind.info/problems/dna/

And have a go at writing the corresponding unittest for this framework using the skeleton provided in **test_dna_count_skeleton.py**

As required by Test-Driven development principles your tests need to capture edge cases, common cases, constraints and unallowed cases to ensure that the code performs correctly in as many scenarios as you can think of.

**Step 2:**

Take a look (you can also run it by inputting some dna strings) at the attempted implementation of the function in **dna.py** called **get_dna_count_first_attempt()** what mistakes do you see? Does your unit tests capture all of them

**Step 3:**

Compare your unit tests with the ones provided in **test_dna_count.py**, how many did you miss? Use them to identity the errors in **get_dna_count_first_attempt()** by running the command:
**>** python3 -m unittest test_dna_count.py

**Step 4:**

Think about how you will fix the code for **get_dna_count_first_attempt()** to make it pass all the tests and implement these changes then run the tests again to confirm that your fix passes all the tests. You can see the correct solution as **get_dna_count_correct_attempt()** in **dna_correct.py**

**Step 5:**

The correct attempt passes all the tests so is a perfectly valid implementation of the specification provided by the domain expert. However correctness is only one criteria (but

arguably the most important) that transformations in your pipeline need to adhere to, another criteria would be computational performance.

From a performance perspective a problem with this implementation is that it uses if statements as it iterates through each character of the dna sequence which is very inefficient as it is best-practice in software engineering to make use of standard library functions that manipulates string whenever possible as they are usually highly optimized. To get a sense of how slow this implementation is, profile it using the time library in python.

**Step 6:**

Now that we have established a correct baseline for this function we can start experimenting in ways of optimizing it without worrying of introducing regression errors due to the availability of thorough unit tests. To optimize this function research some of the string manipulation functions from the python standard library to identify any suitable one for this function.

**Step 7:**

Now implement the function using the string **count()** function, and ensure it is implemented correctly using the implemented unit tests. You can see a correct implementation as the **get_dna_count_optimised()**

Now profile it and compare its performance to get_dna_count_correct_attempt(), not the very notable gain in speed due to using the standard library function.

**Further practice:**

To gain further practice in implementing unit tests and the test-write-run cycle followed in the above 7 steps you can attempt to implement any of the other bioinformatics problem in the Rosalind list using the same workflow (https://rosalind.info/problems/dna/)

**Bonus:**

One of the promises of code assistance powered by Large Language Model AI is that they help boost programmers productivity, especially in writing test code or using test code to generate working prototype.

One way to verify how promising such code assistance is to first provide an LLM model such as chatGPT with the original description of the Rosalind challenge that we solved and ask it to generate unit tests in python. How do you rate its output compared to ours?

Now provide ChatGPT with the unit tests we implemented and the description and then ask it to generate working code in python that passes the tests, how do you rate its output?

Note that we can't read much from this particular tests as the Rosalind list of problems are quite famous and have been on the web for a long time with many correct attempted solutions posted, so it is likely that most LLMs have already been trained on such problems, so to make this a more interesting experiment repeat it by providing it with problem descriptions and unit tests from your own pipeline.