Senior Backend Engineer - Coding Exercise

Our Goal

The goal of this exercise is to test your coding and problem solving skills. When reviewing your submitted code, we'll examine things like the overall strucuture of the code, good seperation of responsibilities for the various code units, and of course the accuracy of your solution.

Most importantly - we want you to succeed! If you find a mistake in our exercise, if anything is unclear, or if you just have some questions about the exercise - please do ping your recruiting manager so that we can help.

Submission Instructions

We expect this coding exercise to take between 2-4 hours, but not more.

1. Please create a dedicated GitHub repository for your code, and send us the link to the repository once you're done.
2. The first commit indicates your **Start Time** *(This can simply be an initial commit with a README.md file that says "First Commit")*.
3. The last commit indicates your **Finish Time**.
4. Your submission should include a README.md file with some instructions on how to run your code, if there's anything you didn't finish, other considerations you thought of during the exercise etc..

Prerequisites

1. You'll need a GitHub account, and to be able to commit and push your changes onto a new GitHub repository.
2. You'll need an IDE with either Python/PIP or NodeJS/NPM setup so that you can develop easily in your favorite language.
3. Make sure to have all these prerequisites ready before you start the test so that you don't lose time on things like "local env setup" (:

The Exercise

At Twist, #We Make DNA.
This exercise requires that you build a DNA Scoring Service.

What does it mean to score DNA?

Every DNA that a customer wishes to synthesize, is first tested for manufacturing difficulty - this is called gene scoring.

The goal is to predict whether or not Twist is going to be able to synthesize the DNA before it gets to our production facility.

The Scoring Service

Build an API service that receives an array of gene sequences, scores the genes *(based on the Scoring Rules listed below)*, and returns a JSON response with the scoring results for each gene sequence.

Input Format

1. The request payload will be in JSON
2. The number of gene sequences in a single request may be in the thousands
3. Each gene sequence is:
    1. A string, whose length is anywhere between 300 and 5000
    2. May includes any permutation of the characters "A", "T", "G", "C"

Input Example

```
[
    "ATGCATGCATATGCGCAATTGCGC...",
    "AAGGTTGGGJJJJJFJAJTJCNAKTTIUANAKAGUTN...",
    ...
]
```

Input Validation

1. For each gene sequence, validate that it only includes the expected characters "A", "T", "G", "C"
2. For each gene sequence, validate that the sequence length is no less than 300, and no more than 5000
3. For the entire payload, validate that there aren't any duplicate gene sequences

Output Format

1. The response payload should be in JSON
2. The response should include a status, message, and the actual results
3. The results should be an array with the scoring result *("valid" or "invalid")*.

Output Example

```
{
    status: "complete",
    message: "gene scoring finished successfully",
    results: [
        "valid",
        "invalid",
        ...
    ]
}
```

Scoring Rules

1. For each gene sequence, the "GC Ratio" must be over 25%, but less than 65%.
   The "GC Ratio" is calculated by taking the total number of "G" characters plus the total number of "C" characters, divided by the length of the entire sequence.
   For example, in the gene sequence "AATTGGCU" there are 2 "G" characters and 1 "C" character, and the length of the entire sequence is 8. Dividing 3 by 8 results in 0.375, so this sequence is valid.

2. **Bonus:** For each gene sequence, calculate the "GC Ratio" of all the sub-sequences *(sub-sequence is 100 characters long)*. Then, make sure that the difference between the highest "GC Ratio" and the lowest "GC Ratior" does not exceed 52% *(if this rule fails, the sequence is invalid)*.

General Guidelines

1. Scale - how will the service handle/accept very large payloads *(gigabytes)*?
2. Tests - is your code testable? *(adding tests isn't a must - but writing testable code is)*
3. Design - make sure your code properly separates the various concerns and responsibilities
4. Start with the simplest solution you have - even if it doesn't fully answer all of the requirements
5. As you iterate over your solution, answer more of the requirements, until you've fulfilled all of them
6. If you didn't have enough time to finish all of the requirements - that's ok! We still want you to submit your results, and we'll review what you have so far *(we're testing your code skills, not your speed)*.