Assignment 03: DNA sequence assembly

Start this assignment early and don't wait until the last minute. You will be writing about the same amount of code (probably less, in fact) than in the last assignment, but it will require more thinking time. It's the kind of thing you might be asked to do (in Python) toward the end of 160. In any event, this is an assignment you may not be able to rush through on a Friday afternoon.

You can (and should!) use Java's built-in `java.util.List` and `java.util.ArrayList` for this assignment. We'll make it clear in future assignments when you're expected to "build your own" data structures (for example: the next one!).

Overview

As you no doubt remember from high school biology, the "code of life" is written in DNA (and its pal, RNA) – triplet "codons" encode a sequence of amino acids, which are assembled into proteins, and so on. An important breakthrough in biological sciences was the ability to replicate DNA in vitro (that is, not in a cell, but in a test tube) using the polymerase chain reaction (PCR). PCR creates many copies of fragments of a sequence of DNA. The equipment for PCR is now cheap enough that some high schools have PCR labs. PCR has many applications. For example, in the *extremely hypothetical* case of a novel coronavirus causing a worldwide pandemic, you might use PCR tests to determine whether a person was infected with such a virus.

Another breakthrough was the ability to *assemble the sequence* of fragments into a coherent whole, thus determining the genetic code (the genome) for an entire organism. For example, the Human Genome Project has sequenced the human genome.

In this assignment, you'll solve a simplified version of the DNA sequence assembly problem, using a simple "greedy" algorithm. That is, given a collection of two or more overlapping DNA fragments, you'll align and merge these fragments by choosing the best matches at each step, with the goal of ending with a single longer sequence.

We've provided a set of unit tests to help with automated testing, though you might also want to write a class with a `main` method for interactive testing. As before, we've disable the timeout code so you can use the debugger, but if your code gets stuck during testing, you might want to uncomment these two lines at the top of each test file:

```
@Rule
public Timeout globalTimeout = Timeout.seconds(10); // 10 seconds
```

Goals

- Translate written descriptions of behavior into code.
- Practice writing instance methods, including overriding methods of `Object`.
- Practice interacting with the `List` abstraction.

- Test code using unit tests.

Downloading and importing the starter code

As in previous assignments, download and decompress the provided archive file containing the starter code. Then import it into Code in the same way; you should end up with a `dna-sequence-assembly-student` project.

What you will be doing

The most important thing to understand is how you'll assemble shorter overlapping fragments into longer ones.

First, what does a fragment look like? It's a sequence of one or more nucelotides; each is one of adenine (A), cytosine (C), guanine (G) and thymine (T). So a fragment might look like one of

```
CGCAT
CATGAC
ACATG
```

Next, how do we assemble them? We're going to use a simple greedy algorithm, which, quoting Wikipedia, reads as follows:

> Given a set of sequence fragments the object is to find the shortest common supersequence.

1. Calculate pairwise alignments of all fragments.
2. Choose two fragments with the largest overlap.
3. Merge chosen fragments.
4. Repeat step 2 and 3 until only one fragment is left.

Let's look at the first few steps in more detail, since they're the least clear.

Calculating pairwise alignments

What's a pairwise alignment (and what's its overlap)? Let's look at an example. Consider our first two fragments, listed above: `CGCAT` and `CATGAC`. We can see how much their "ends" overlap if we put `CGCAT` first and then `CATGAC`:

```
  these three overlap
  vvv
CGCAT
  CATGAC
  ^^^
  these three overlap
```

Here, there's an overlap of three (`CAT`). If we were to merge these together, the result would be `CGCATGAC`:

```
 CGCAT
+  CATGAC
---------
 CGCATGAC
```

If we tried them the other way around, what would the overlap and merged fragment look like?

```
 CATGAC
+     CGCAT
-----------
 CATGACGCAT
```

Here the overlap is only one and the result would be `CATGACGCAT`.

You should now see (1) that any two fragments can have an overlap of at least zero and at most the length of the shorter fragment, and (2) that order matters when comparing overlaps: the front of one fragment can be checked against the rear of another, but that's different from checking the rear of the first against the front of the second.

Finally, note that we will only consider overlaps on the end, and not worry about one fragment being entirely embedded within another. That is, your code **must not** check for things like:

```
 GCTCAGC
+  TCA
--------
 GCTCAGC
```

Though two identical fragments will be merged, as they are only compared on the end. In other words, we *do* expect you to merge fragments like:

```
 GCTCAGC
+GCTCAGC
--------
 GCTCAGC
```

Choosing the largest overlap

Given a collection of fragments, you can compare every fragment against every other fragment (in both orders) and find the pair with the largest overlap. What do we mean by both orders? Consider each fragment as both a left fragment against every other on its right, and a right fragment against every other on its left.

But what if two have the same overlap? I want you to break ties by choosing the pair whose merger results in the shorter merged sequence.

If there are further ties, do what you like — I will make sure there are no tests that are ambiguous, and I don't want your `merge` method to be sixteen special cases long.

(In a practical sequence assembler, deciding how to handle ambiguity is very important, as are many other cases: What about "almost perfect" matches, as real PCR occasionally induces errors in the fragments? Or subsets, which I told you to ignore? Or how little overlap is so little as to be not worth merging? And so on. But we won't worry about those details here.)

Merging the fragments

Suppose we are still working with our example three fragments, `CGCAT`, `CATGAC`, and `ACATG`. Further suppose they're stored in a list, which we'll write as [`CGCAT`, `CATGAC`, `ACATG`].

If, after checking, we decided to merge the first two (as described above), our list would look like: [`CGCATGAC`, `ACATG`]. Then we'd merge again and be left with a single entry in our list: [`CGCATGACATG`].

What to do

As usual, look over the files we've provided. The `Fragment` class represents a single fragment; the `Assembler` class keeps a list of `Fragments` and assembles them into longer `Fragments`.

Start with the `Fragment` class. Here are some hints to get you started there:

- You'll need to store the nucleotide sequence in an instance variable. `String` is probably the easiest thing to use.
- `length` and `toString` should be straightforward, if you used a `String` to store the nucleotides.
- You'll need to write an `equals()` method to compare the current (`this`) `Fragment` to another `Fragment o`. Use Visual Studio Code to do this for you, on the basis of your instance variable. Do not be alarmed that **many tests will fail until you implement the `Fragment.equals()` method** – not because your other code is wrong, but because the tests use `equals`. It's how `assertEquals` checks for equality on objects.
- Look over the instance methods of `String` when writing `calculateOverlap` and `mergedWith`. In particular, you might find `startsWith`, `endsWith`, and `substring` helpful. Try breaking the solution up into conceptual chunks.
- One such chunk to consider: You might add a new method `boolean hasOverlap(Fragment f, int overlapLength)` that checks (that is, returns `true`) if the current `Fragment` overlaps with another fragment `f` with an overlap of `overlapLength`. Then use it to implement `calculateOverlap` by checking iteratively checking for a maximum-size overlap, then one less, then one less, etc., until you find the largest overlap.
- If you choose to write a `hasOverlap` method, you might want to add some tests. For example:

```
@Test
public void testHasNoOverlap() {
    Fragment f = new Fragment("GCAT");
    Fragment g = new Fragment("CGTA");
```

```
        assertFalse(f.hasOverlap(g, 1));
        assertFalse(g.hasOverlap(f, 1));
    }

    @Test
    public void testHasSomeOverlap() {
        Fragment f = new Fragment("GGGA");
        Fragment g = new Fragment("AGGG");
        assertTrue(f.hasOverlap(g, 1));
        assertFalse(f.hasOverlap(g, 2));
        assertTrue(g.hasOverlap(f, 1));
        assertTrue(g.hasOverlap(f, 2));
        assertTrue(g.hasOverlap(f, 3));
        assertFalse(g.hasOverlap(f, 4));
    }
```

Once you have `Fragment` passing the tests, start on `Assembler`. Again, some hints:

- The constructor and `getFragments` should be straightforward, though note the copy requirement in the constructor – your constructor must make and store a copy of the list of fragments – it should *not* modify the original list, or reference it except to make a copy! You can use the "copy constructor" of `ArrayList` to do this. In particular, if you pass a list as an argument to the `ArrayList` constructor, it returns a copy of the list you passed in. (Specifically: something like `this.fragments = new ArrayList<Fragment>(fragments);`)
- For `assembleOnce`:
  - (Note you may not need this hint, depending upon how you structure your code.) Sometimes we use `-1` or `0` as the initial value of a variable that we're checking against to track a maximum. What if you want to initialize a variable that's tracking a minimum? Use `Integer.MAX_VALUE` in this case.
  - You're probably going to need to write a nested `for` loop (that is, a `for` loop inside a `for` loop) to check each pair of fragments. Remember not to compare a fragment against itself (and think about whether this check should be using `==` or `equals`).
  - Remember to add the newly created merged `Fragment` to the list, and to remove from the list the two `Fragments` that were merged.
- `assembleAll` will be a one- or two-liner once you get `assembleOnce` working – just call `assembleOnce` repeatedly until it returns false.

Submitting the assignment

When you have completed the changes to your code, you should export an archive file containing the entire Java project. To do this, follow the same steps as from Assignment 01 to produce a `.zip` file, and upload it to Gradescope. Note that if you want things to upload faster, you can use an external program to zip only the `src/` directory by expanding the project; that's all this autograder requires.

Remember, you can resubmit the assignment as many times as you want, until the deadline. If it turns out you missed something and your code doesn't pass 100% of the tests, you can keep working until it does.