TL;DR

Want a quick overview of what to do with no details? http://bit.ly/201spring19-dna-tldr

Project Introduction

BackGround

This background is interesting, but not really needed to do the assignment. There are some good stories here, but if you want to get to the assignment, you can skip this stuff.

In this assignment you'll experiment with different implementations of a simulated *restriction enzyme* cutting (or cleaving) a DNA molecule. Three scientists shared the Nobel Prize in 1978 for the discovery of restriction enzymes. They're also an essential part of the process called PCR *polymerase chain reaction* which is one of the most significant discoveries/inventions in chemistry and for which Kary Mullis won the Nobel Prize in 1993.

Kary Mullis, the inventor of PCR, is an interesting character. To see more about him see this archived copy of a 1992 interview in Omni Magazine or his personal website which includes information about his autobiography *Dancing Naked in the Mind Field*, though you can read this free Nobel autobiography as well.

The simulation is a simplification of the chemical process, but provides an example of the utility of linked lists in implementing a data structure. The linked list code you'll write and reason about is an example of a chunk list.

Git

Fork, clone, and import the cloned project from the file system. Use this URL from the course GitLab site: https://coursework.cs.duke.edu/201spring19/dna-link-spring19 ***. Be sure to fork first*** (see screen shot). Then Clone using the SSH URL after using a terminal window to cd into your Eclipse workspace. Recall that ***you should import using*** the File>Import>General>Existing Projects into Workspace -- then navigate to where you cloned the diyad project.

DO NOT DO NOT import using the Git open -- use General>Existing Projects Into Workspace.

Pushing to Git

When you make a series of changes you want to 'save', you'll push those changes to your GitHub repository. You should do this after major changes, certainly every hour or so of coding. You'll need to use the standard Git sequence to commit and push to GitHub:

```
git add .
```

```
git commit -m 'a short description of your commit here'
git push
```

Partners

You may work with a partner ***from your discussion section*** on this assignment. One person should fork-and-clone from the GitLab repo. That person will add the other person/partner as a collaborator on the project. For full information, see the documentation here: https://docs.gitlab.com/ee/user/project/members/

Choose Settings>Members>Invite Members. Then use the autocomplete feature to invite your partner to the project. Both of you can clone and push to this project.

1. First, one person should create the GitLab repository then add the partner as a maintainer to the project.
2. Both students should clone the same repository and import it into Eclipse.
3. After both students have cloned and imported, one person should add a comment to the **`LinkStrand.java`** class with their name in a comment at the start of the file. Commit and push this change.
4. The other partner will then use the command line and issue a **`git pull`** request. Simply use the command-line and type:

   ```
   git pull
   ```

5. After this command, right-click on the project in Eclipse and choose the Refresh menu item. You should see the modified **`LinkStrand.java`** file with a new comment. Add your name in a comment, then commit and push. The other person will need to issue a **`git pull`** to get that file.

As long as partners are modifying different files, this process works seamlessly. Modifying the same file can lead to issues in resolving conflicts. Git will deal with this with your help, but it's better to take turns in working on the same file, or to work on different files within the project. ***Ideally you'll always be physically together when working on the project.***
When submitting you'll use the partner/team in Gradescope. That's described below in the submission section.

Can't find partner in discussion? Request a partner: http://bit.ly/201spring19-dna-partnerrequest

Overview of Programming

You'll run benchmarks for three classes, **`StringStrand`**, **`StringBuilderStrand`** and **`LinkStrand`**. They are different implementations of the **`IDnaStrand`** Interface. The first two are provided in the starter code and you'll complete the implementation of the class

`LinkStrand`. You'll also design, develop, and run benchmark classes as part of the analysis section.

This assignment has two parts: Part 1 (due before Spring Break) consists of cloning the starter repo and answering Questions 1 and 2 of the analysis only: No code is required for this part. Part 2 (due after Spring Break) consists of writing the code for the `LinkStrand` and `CodonProfiler` classes, and then answering Question 3 of the analysis.

There are four components to this assignment:

1. **[Part 1]** Run benchmarks on two provided classes. Describe the results of running the benchmarks in your analysis.txt file. The description of what to do for running the benchmarks is described in the Benchmark Results section. You can run two of the three benchmarks before completing the implementation of **`LinkStrand`** described next. ***Complete this before Spring break, no programming.***
2. **[Part 2]** Implement **`LinkStrand`** that extends the **`IDnaStrand`** interface so that it is correct and meets performance requirements. Test and benchmark this class and report on these tests in your analysis.txt file.
3. **[Part 2]** Change the implementation of **`CodonProfiler.getCodonProfile`** so that it runs in **`O(N)`** time instead of **`O(C×N)`** time for an array of **`C`** codons and a strand of **`N`** characters.
4. **[Part 2]** Answer questions about the benchmark code. See the Analysis section for details.

`LinkStrand` implements `IDnaStrand`

Create a new class named **`LinkStrand`**. This class must implement the **`IDnaStrand`** interface as explained in some detail below. You should allow Eclipse to fill in all the methods needed to implement the interface with stub interfaces, e.g., that return **`null`** or zero for example. Then you'll implement and test the methods as described here and in the section that follows on how the class works. Here's the header for the class that you'll implement:

```
public class LinkStrand implements IDnaStrand
```

If you let Eclipse add the unimplemented methods you'll see **`@Override`** with seven methods that are described below.

You'll implement two constructors as described below. The constructors and methods don't need to be implemented in the order shown, but the simpler methods are listed first. These methods are tested in the **`TestStrand`** class except for **`charAt`** which is tested in the **`TestIterator`** class. In descriptions below **`N`** is the number of nucleotides/basepairs/characters in a strand.

1. Implement two constructors: ***one with no parameters (the default constructor)*** and one with a **`String`** parameter.
    a. The constructors work by calling the required **`initialize`** method, see **`StringStrand`** for an example.
2. Implement the **`initialize`** method that initializes the **`LinkStrand`** object with a **`String`**.
3. Implement the **`getInstance`** method that works similarly to what you see in **`StringStrand`** and **`StringBuilder`** strand. ***This must return a `LinkStrand` object.***
4. Implement **`size`**. This should be a single line and must run in **`O(1)`** time.
5. Implement **`getAppendCount`**. This should be a single line and must run in **`O(1)`** time.
6. Implement **`toString`**. This returns the **`String`** representation of the **`LinkStrand`** by looping over nodes and appending their values to a **`StringBuilder`** object. The method should run in **`O(N)`** time.
7. Implement **`append`** which creates one new node and updates instance variables to maintain class invariants as described below.
8. Implement **`reverse`** to return a new **`LinkStrand`** object that's the reverse of the object on which it's called. ***This method is not a mutator, it creates a new `LinkStrand`.***
9. Implement **`charAt`** which returns the character at a specific index. This method requires new instance variables to meet performance characteristics.

You should test each method as you implement it using the **`TestStrand`** JUnit test class. You'll need to change the type of strand returned in that JUnit class method **`getNewStrand`** to test your class***. It's unlikely that any tests will work until you've implemented `LinkStrand.toString()`.***

`LinkStrand` State, Constructors and **`initialize`** method

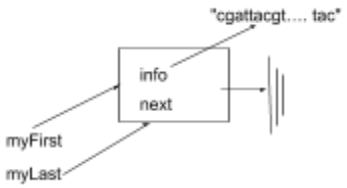
You should start with the following definitions for a private inner class and instance variables to use a linked-list internally as part of the **`LinkStrand`** class. ***Note that all are private***.

```
private class Node {
  String info;
  Node next;
  public Node(String s) {
       info = s;
       next = null;
  }
}
private Node myFirst,myLast;
private long mySize;
```

```
private int myAppends;
```

All constructors and methods must maintain the following class invariants:

1. **`myFirst`** references the first node in a linked list of nodes.
2. **`myLast`** references the last node in a linked list of nodes.
3. **`mySize`** represents the total number of ***characters*** stored in all nodes together.
4. **`myAppends`** is the number of times that the **`append`** method has been called. ***It would be useful to think of this as one less than the number of nodes in the linked list.***



Initially, when the **`LinkStrand("cgatt")`** constructor is called (though the String parameter can be any string) there will be a single **`Node`** in the linked list that represents the DNA strand **`"cgatt"`**. (The only way to have more than one node in a **`LinkStrand`** internal linked-list is by calling **`.append`**.)

As described above, you'll create two constructors. The string constructor should consist of ***one call to*** **`initialize`** which establishes the class invariant with a single node representing the entire strand of DNA as illustrated above. The no-argument constructor, sometimes called the default constructor, should have one line: **`this("")`** which calls the other constructor with a **`String`** parameter of **`""`**.

The **`initialize`** ***method will maintain the class invariants when it's called. There will be a single node created after*** **`initialize`** ***is called.***

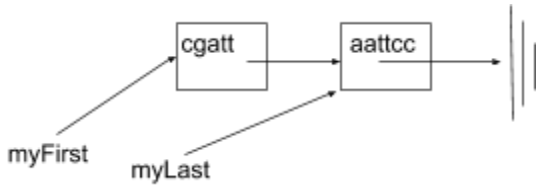
Implementing the `append` Method

The ***`append`*** ***method should add one new node to the end of the internal linked list and update state to maintain the invariant.***

For example, suppose that these two statements are both executed:

```
LinkStrand dna = new LinkStrand("cgatt");
dna.append("aattcc");
```

The internal linked list maintained by **`LinkStrand`** after the first call is diagrammed above. After the call to append we have the following picture:



Note that maintaining the class invariant after this call to append would require

1. **`myFirst`** doesn't change
2. **`myLast`** changes to point to the new node added
3. **`mySize`** is incremented by six
4. **`myAppends`** is incremented by one (because a new node is added).

Note that **`.append`** returns an **`IDnaStrand`** object. This is the object that was just modified/appended to. However, the method **`append`** ***does not*** create a new **`IDnaStrand`** object. The **`.append`** method is a mutator -- it changes the internal state of the **`IDnaStrand`** object on which it's invoked, and then returns this **`LinkStrand`** object itself. Look carefully at both **`StringStrand`** and **`StringBuilderStand`** strand to see what to return.

Note that after implementing **`append`**, the method **`getAppendCount`** should return the correct result, the value of instance variable **`myAppends`** that's maintained by the class invariants and initialized/updated in **`initialize`** and **`append`**.

Implementing the `toString` method

The **`toString`** method returns the **`String`** representation of the entire DNA strand. ***Conceptually this is a concatenation of the `String` stored in each node.***

This method should use a standard **`while`** loop to visit each node in the internal linked list. The method creates and updates a single **`StringBuilder`** object by appending each **`node.info`** field to a **`StringBuilder`** object that's initially empty. The final return from **`LinkStrand.toString`** will simply be returning the result of calling **`.toString()`** on the **`StringBuilder`** object. See the **`DNABenchmark.dnaFromScanner`** implementation for guidance on the **`StringBuilder`** strand class.

For more guidance on **`StringBuilder`**, see the Java Documentation here.

You should be able to test all the methods implemented to this point using the class `TestStrand`. The testing methods in `TestStrand` rely on `.toString` being correct, so after implementing `.toString` you may find errors in your other methods as a result of testing.

Implementing the `reverse` method

This method creates a new **`LinkStrand`** object that is the reverse of the object on which it's called. The reverse of **`"cgatccgg"`** is **`"ggcctagc"`**. ***This method returns a new strand; it does not alter the strand on which it's called, i.e., it's not a mutator.***

Note: you must create N new nodes in reversing a LinkStrand object with N nodes. If you do not, you are likely mutating/changing the LinkStrand being reversed.

You'll need to reverse the linked list, and reverse each string in each node of the linked list. Specifically, the reversed **`LinkStrand`** should have the ***same number of nodes*** as the original **`LinkStrand`**, but in reverse order; each internal node should also contain the reversed **`String`** of the corresponding node in the original **`LinkStrand`**.

To reverse a **`String`** use a **`StringBuilder`** appropriately --- see **`StringStrand.reverse`** for details on using the **`StringBuilder.reverse`** method.

Note that in creating a new linked list that's the reverse of the list of nodes being traversed it's easiest to simply add a new node to the front of the reversed list being constructed. So in reversing a->b->c->d and traversing in order a,b,c,d, your code would have created the list c->b->a after traversing the first three nodes: a,b,c. When reaching the next or 'd' node your code adds 'd' to the front of this list creating d->c->b->a.

Implementing the `charAt` method

This method returns the character at the specified index if that's a valid index, and throws an **`IndexOutOfBoundsException`** otherwise. A naive implementation of this method would start at the beginning of the linked list, the node referenced by **`myFirst`** and count characters until the index-th character is found.

For full credit (and to pass the timing tests in `TestIterator`) you'll need to maintain state so that after a call of `charAt(k)` the call of `charAt(k+1)` is an `O(1)` operation. This will make the loop below **`O(N)`** for an N-character strand. See the next section for help on this.

Basic, Correct but Inefficient Implementation of `charAt`

First we'll show an inefficient implementation of the **`charAt`** method --- a method to find a character at a specific index in a linked list of strings. Your code will need to traverse the linked list counting characters. ***The code below illustrates how to do this. It doesn't check to see if parameter index is valid, but it passes the JUnit tests for correctness.***

```
public int charAt(int index) {
     int count = 0;
```

```
        int dex = 0;
        Node list = myFirst;
        while (count != index) {
              count++;
              dex++;
              if (dex >= list.info.length()) {
                    dex = 0;
                    list = list.next;
              }
        }
        return list.info.charAt(dex);
    }
```

This code will pass correctness tests, e.g., in **`TestIterator`**. However, it's not efficient since it starts at the beginning of the linked list for each call.

You should be sure you understand how local variables `count` and `dex` are used in the code above before trying to make the code more efficient for a sequence of calls as explained in the next section.

Efficient Implementation of **`charAt`**

You should create instance variables in the class `LinkStrand` so that after a call of `charAt(k)`, calling `charAt(k+1)` is an `O(1)` operation.

You can see in the code above that three values are used, these correspond to the three variables shown above:

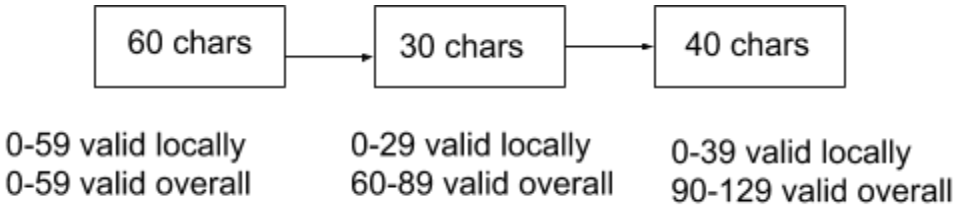
- the current node in the iteration (**`list`**),
- the current index in that node (**`dex`**),
- the overall count (**`count`**).

You should ***create and initialize three corresponding instance variables***: one for the current node in a sequence of calls of **`charAt`**, one for the current index into that node, and one for the overall count. You'll need to initialize these instance variables when a strand is created, e.g., in method **`initialize`**. You'll use these instance variables on each call to **`charAt`**, and you'll update them similarly to how the local variables are updated -- to save the current “progress” so that on a subsequent call to **`charAt`** it might be possible to continue from the last call and access the next character in a node, for example, by simply incrementing the index into that node. Of course this might require going to the next node if the current node is "out" of characters just as is done in the code shown above.

Use these three instance variables and update them appropriately in **`charAt:`**

- **`myIndex`** is the value of the parameter in the last call to **`charAt`**. This means that if a call to **`s.charAt(100)`** is followed by **`s.charAt(101)`** the value of **`myIndex`** will be 100 after **`s.charAt(100)`** executes and 101 after **`s.charAt(101)`** executes.

- **`myLocalIndex`** is the value of the index within the string stored in the node last-referenced by **`charAt`** when the method finishes. For example, suppose a strand consists of three nodes: the first has 60 characters; followed by a node of 30 characters; followed by a node of 40 characters. The call **`s.charAt(40)`** will mean that **`myIndex`** is 40 and **`myLocalIndex`** is also 40 since that's the index within the first node of the list,



  where the character whose index is 40 is found. Suppose this is followed by **`s.charAt(70)`**. The character at index 60 of the entire strand will be the character with index zero of the second node -- since the first node holds characters with indexes 0-59 since its info field is a string of 60 characters. The character at index 70 of the entire strand will be the character with index 10 of the second node.
- **`myCurrent`** is the node of the internal list referenced in the last call to **`charAt`**. In the example above the value of **`myCurrent`** would be the first node after the call **`s.charAt(40)`**, would be the second node after the call **`s.charAt(70)`** or **`s.charAt(89)`**, and would be the third node after the call **`s.charAt(90)`** since the first two nodes only contain a total of 90 characters, with indexes 0 to 89.

In the **`TestIterator`** code you get with this assignment, there are correctness tests and performance tests for going forward in **`O(N)`** time as described here.

Why do we need **`charAt`** to be efficient?

If the **`charAt`** method is not efficient, the loop below will be **`O(N`**^{2}**`)`** since the **`charAt`** method will be **`O(k)`** to access the kth character.

```
LinkStrand dna = new LinkStrand(".....");
StringBuilder s = new StringBuilder("");
for(int k=0; k < dna.size(); k++) {
    s.append(dna.charAt(k));
}
```

This **`charAt`** method is called by the code in the **`CharDnaIterator`** class. So iterating over an IDnaStrand object will ultimately use the **`charAt`** method as shown in the code below.

```
LinkStrand dna = new LinkStrand(".....");
Iterator<Character> iter= dna.iterator();
for(char ch : iter) {
    System.out.print(ch);
```

```
        }
        System.out.println();
```

The `Iterator` object in the code above is constructed as a result of calling the default `IDnaStrand.iterator` method, the body is shown here:

```
    return new CharDnaIterator(this);
```

Note that the `IDnaStrand` object referenced by `this` is then stored in the `CharDnaIterator` object being created.

You only need to implement charAt, then all the code described and shown above will work correctly! You will need to initialize the instance variables too.

Order of Calls Matters

However, you'll need to write code to deal with calls that aren't "in order". If the call `.charAt(100)` is followed by the call `.charAt(30)` you'll need to start at the beginning of the internal linked list to find the character with index 30. If `.charAt(100)` is followed by `.charAt(350)` you won't start at the first node, but continue with the values stored in the instance variables.

`CodonProfiler`

The class you're given has a method that determines how many times each 3-character codon in an array of codons occurs in a strand of DNA. You should look at the implementation of `getCodonProfile` and the tests in `CodonProfileTest` that run the code to see what's expected. The code you're given runs in `O(C×N)` time for an array of `C` codons and a stand of `N` characters. Verify this by looking at the code. The outer loop goes every each codon, and for each one the entire strand is searched via an iterator to find how many times that codon occurs.

Replace this implementation using a `HashMap` so that the code runs in `O(N)` time.

You should count how many time every possible codon in the strand occurs using the map. After looking at every possible codon in the strand, and counting each one regardless of whether it is in the array `codons`, write code to loop over the values `codons` to create the array to return. Technically this will be `O(N+C)` since you must loop over all the codons after creating the map.

Benchmark Results

You'll need to run the `DNABenchmark` class three times, once for each implementation of the IDnaStrand interface: `StringStrand`, `StringBuilderStrand`, and `LinkStrand`. You

change the runs by changing the value of the static instance variable **`strandType`** at the top of the class file. ***You should copy/paste the output that's generated by running the benchmark program using the large*** **`ecoli.txt`** ***file that's in the data folder you get when the project is cloned. The benchmark runs until memory is exhausted.*** Results are shown from an instructor/TA laptop. ***You should generate your own results from the machine you run the benchmark code on. The*** **`StringStrand`** ***class will take a very long time to run!***

Your results will not be the same as those shown here, and you may run out of memory with more or fewer experiments. But you should copy/paste the results of your runs into your **`analysis.txt`** file and explain the results (see the Analysis section for details).

Note that the benchmark code runs two experiments to average the results. So the time for **`StringStrand`** in real time is much longer than what's reported as the average (at least double, then more).

Benchmark for StringStrand

These results take a long time to run before exhausting memory. ***Anticipate more than 15 minutes.*** Generated running **`DNABenchmark`** with **`strandType`** to **`"StringStrand"`**

```
dna length = 4,639,221
cutting at enzyme gaattc
-----
Class                  splicee          recomb    time appends
-----
StringStrand:              256       4,800,471 0.421 1290
StringStrand:              512       4,965,591 0.394 1290
StringStrand:            1,024       5,295,831 0.420 1290
StringStrand:            2,048       5,956,311 0.479 1290
StringStrand:            4,096       7,277,271 0.587 1290
StringStrand:            8,192       9,919,191 0.822 1290
StringStrand:           16,384      15,203,031 1.290 1290
StringStrand:           32,768      25,770,711 2.207 1290
StringStrand:           65,536      46,906,071 4.455 1290
StringStrand:          131,072      89,176,791 9.741 1290
StringStrand:          262,144     173,718,231 22.162     1290
StringStrand:          524,288     342,801,111 44.144     1290
StringStrand:        1,048,576     680,966,871 82.143     1290
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
     at StringStrand.append(StringStrand.java:70)
     at IDnaStrand.cutAndSplice(IDnaStrand.java:41)
     ...
```

Benchmark for StringBuilderStrand

Changing **`strandType`** to **`"StringBuilderStrand"`** yields the results below on an instructor laptop. Memory runs out much more quickly, in seconds, but with the same size as **`StringStrand`**.

```
dna length = 4,639,221
cutting at enzyme gaattc
-----
Class                 splicee          recomb    time appends
-----
StringBuilderStrand:              256       4,800,471 0.026 1290
StringBuilderStrand:              512       4,965,591 0.020 1290
StringBuilderStrand:            1,024       5,295,831 0.007 1290
StringBuilderStrand:            2,048       5,956,311 0.006 1290
StringBuilderStrand:            4,096       7,277,271 0.006 1290
StringBuilderStrand:            8,192       9,919,191 0.009 1290
StringBuilderStrand:           16,384      15,203,031 0.010 1290
StringBuilderStrand:           32,768      25,770,711 0.022 1290
StringBuilderStrand:           65,536      46,906,071 0.031 1290
StringBuilderStrand:          131,072      89,176,791 0.069 1290
StringBuilderStrand:          262,144     173,718,231 0.159 1290
StringBuilderStrand:          524,288     342,801,111 0.402 1290
StringBuilderStrand:        1,048,576     680,966,871 0.631 1290
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
     at java.base/java.util.Arrays.copyOf(Arrays.java:3744)
     at
java.base/java.lang.AbstractStringBuilder.ensureCapacityInternal(Abstr
actStringBuilder.java:146)
    ...
```

Benchmark for LinkStrand

Generated running **`DNABenchmark`** with **`strandType`** to **`"LinkStrand"`** after that class has been tested. Note that this takes just seconds to run, and strand/splicee sizes are significantly larger than with the other **`IDnaStrand`** types.

```
dna length = 4,639,221
cutting at enzyme gaattc
-----
Class                 splicee          recomb    time appends
-----
```

```
LinkStrand:                256        4,800,471      0.028 1290
LinkStrand:                512        4,965,591      0.023 1290
LinkStrand:              1,024        5,295,831      0.005 1290
LinkStrand:              2,048        5,956,311      0.003 1290
LinkStrand:              4,096        7,277,271      0.004 1290
LinkStrand:              8,192        9,919,191      0.006 1290
LinkStrand:             16,384       15,203,031      0.004 1290
LinkStrand:             32,768       25,770,711      0.003 1290
LinkStrand:             65,536       46,906,071      0.004 1290
LinkStrand:            131,072       89,176,791      0.004 1290
LinkStrand:            262,144      173,718,231      0.003 1290
LinkStrand:            524,288      342,801,111      0.004 1290
LinkStrand:          1,048,576      680,966,871      0.006 1290
LinkStrand:          2,097,152    1,357,298,391      0.004 1290
LinkStrand:          4,194,304    2,709,961,431      0.006 1290
LinkStrand:          8,388,608    5,415,287,511      0.003 1290
LinkStrand:         16,777,216   10,825,939,671      0.003 1290
LinkStrand:         33,554,432   21,647,243,991      0.006 1290
LinkStrand:         67,108,864   43,289,852,631      0.005 1290
LinkStrand:        134,217,728   86,575,069,911      0.006 1290
LinkStrand:        268,435,456  173,145,504,471      0.005 1290
LinkStrand:        536,870,912  346,286,373,591      0.004 1290
LinkStrand:      1,073,741,824  692,568,111,831      0.005 1290

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
     at java.base/java.util.Arrays.copyOf(Arrays.java:3744)
    ...
```

JUnit Tests

You should run the JUnit tests in both **`TestStrand`** and **`TestIterator`**. Verify that these work for **`StringStrand`** and **`StringBuilderStrand`** and then use these classes to test your **`LinkStrand`** implementation. You'll use the **`CodonProfilerTest`** to test the code you write in **`CodonProfile`**.

Analysis

Answer three questions in your analysis.txt file. The questions are here: http://bit.ly/201spring19-dna-analysis ***-- The document includes detailed explanations and diagrams, be sure to read that and to not only rely on on what's in analysis.txt.***

For DNA Part 1, due before Spring Break, you'll only submit the answers to the first two analysis questions on **`StringStrand`** and **`StringBuilderStrand`** classes. You'll push your project to Git and then Gradescope, but only the analysis.txt file will be looked at by the UTAs. ***You do not need to write any code for this part.***

For DNA Part 2, due after Spring Break, you'll write the code for **`LinkStrand`** and answer the third question.

After completing the analysis questions you should push to Git and submit the entire project on Gradescope. Please note that Analysis is part of the assignment, and Gradescope is the only portal to submit the assignment, including the analysis. We reserve the right to reject your analysis submission if you forget to upload it to Gradescope, even if it's in your Git repo.

Submitting

You'll submit the code to Gradescope after pushing your program and **`analysis.txt`** file to GitLab. This is a partner/team project in Gradescope, so one person will be able to add the other as partner if you're working together. ***Only one person a group submits, not both.***

Refer to this document for submitting to Gradescope with a partner:

Reflect

You can access the reflect form at http://bit.ly/201spring19-dna-reflect

You should NOT complete the reflect form until you're done with all the coding portion of the assignment. Since you may uncover bugs from the autograder, you should wait until you've completed debugging and coding before completing the reflect form.

Grading

Points	Grading Criteria
16	Code for LinkStrand. This includes 4 for correct and efficient implementation of .charAt method, as well as 2 for API ***and*** Javadoc comments in LinkStrand.
4	Implementation of CodonProfiler that runs in O(N) time and is correct.
8	Analysis code and questions answered. UTAs will grade and comment on this. Part 1 is 6 points, part 2 is 2.

2	Comments and code style. UTAs will grade and comment on this.

See this piazza post for expectations of Javadoc comments. ***After writing the comments you should push to Git and submit the entire project on Gradescope.***

We will map total points you earn to scores as follows. This means if you lose five points, you receive the same score as losing zero points: an A.

```
25-30:  A
19-24:  B
13-18:  C
 8-12:  D
```