C212 Final Exam (130 points)
December 2023

Please read these directions before starting your exam.

This is a closed-note exam aside from your one page of notes, double-sided. You may not use any electronic devices to complete this exam, nor can you communicate with anyone besides the proctors and professor. *If you are caught cheating, you will receive an F in the course.*

For any question, unless specified otherwise, you may use any class without a corresponding `import`. E.g., if you want to use `HashMap`, you do not need to also import `java.util.HashMap`.

Unless otherwise stated, you do not need to spell out the "full design recipe", i.e., write the signature, documentation comments, and tests. Of course, doing so may aid you in your solution.

If you find a mistake, please raise your hand and let one of the proctors know; we will determine whether or not this is the case.

When you are finished, turn in your exam and notes sheet if you have one, then quietly exit.

You have 120 minutes to complete the exam.

Good luck!

Question	Points	Score
1	60	
2	30	
3	20	
4	20	
Total:	130	

Name: ______________________________

IU Email: ______________________________

1. (60 points) In functional programming languages, we often use three operations to act on data structures akin to linked lists: *cons*, *first*, and *rest*. In this question, you will implement a *cons*-like data structure, since Java has no real equivalent.

   We can define a *cons* list as follows:

```
A ICons is one of:
  - EmptyConsList
  - new ConsList(x, ICons)
```

   We need a way of linking these two types together, i.e., `EmptyConsList` and `ConsList`. So, we will design an interface `ICons<T>` to hook the two together.

```
interface ICons<T extends Comparable<T>> extends Comparable<ICons<T>> {

  /**
   * Retrieves the first element of the list.
   */
  T getFirst();

  /**
   * Retrieves the rest of the list, i.e., the list without
   * the first element.
   */
  ICons<T> getRest();

  /**
   * Determines whether this list is the empty list.
   */
  boolean isEmpty();
}
```

(a) *(10 points)* Design the `ConsList<T>` class and the `EmptyConsList<T>` classes. Both classes should implement the `ICons<T>` interface. The former should store two variables: an element of type `T`, and a `ICons<T>` representing the rest of the list. The latter should store no instance variables. **Do not override/implement the methods defined inside `ICons<T>` yet—we will do that in subsequent steps.**

```
class ConsList<T extends Comparable<T>> _________________________ {







}

class EmptyConsList<T extends Comparable<T>> _________________________ {







}
```

(b) *(5 points)* Design a `private` constructor for `ConsList`. It should receive the `first` and `rest` arguments, and assign them to the respective instance variables.

(c) *(3 points)* Design the `public` constructor for `EmptyConsList`.

(d) *(5 points)* Inside `ConsList`, override the `getFirst`, `getRest`, and `isEmpty` methods, which retrieve the respective parts of the list, and determine if the list is empty respectively.

(e) *(5 points)* Inside `EmptyConsList`, override the `getFirst` and `getRest` methods, wherein each method throws an `IllegalOperationException`, since accessing the parts of an empty list is nonsensical. Then, override `isEmpty` as appropriate.

(f) *(7 points)* Inside `ConsList`, write the `setFirst` and `setRest` methods, which respectively mutate the given list instance variables.

(g) *(7 points)* Inside `ConsList`, write the `static ConsList<T> cons(T first, ICons<T> rest)` method, which invokes the `private` constructor and returns a new *cons* list.

(h) *(8 points)* We see that `ICons` extends `Comparable<ICons<T>>`, meaning that it has to provide a definition of `compareTo`. Override `compareTo` in both `ConsList` and `EmptyConsList` to compare the elements of a *cons* list. If, in `ConsList`, the argument is not a `ConsList`, return 1.

(i) *(10 points)* Write coherent tests for your `ICons<T>` data structure. In particular, you should test the following methods: `getFirst`, `rest`, `setFirst`, `setRest`, and `isEmpty`. It might make sense to create a couple of lists outside each test method, then test them inside those methods.

```
import static Assertions.assertAll;
import static Assertions.assertEquals;

class ConsListTester {






  @Test
  void consListGetFirstTest() {


  }

  @Test
  void consListGetRestTest() {


  }

  @Test
  void consListSetFirstTest() {


  }

  @Test
  void consListSetRestTest() {


  }

  @Test
  void consListIsEmptyTest() {


  }
}
```

2. (30 points) This question has five parts. We need to provide some background for the question first. An *encoded string* S is one of the form:

$$S = n[S']$$
$$S' = SS' \mid [a, ..., z]^* \mid \texttt{""}$$

We imagine this didn't clear up what the definition means. Take the encoded string `"3[a]2[b]"` as an example. The resulting decoded string is `"aaabb"`, because we create three copies of `"a"`, followed by two copies of `"b"`. Another example is `"4[abcd]"`, which returns the string containing `"abcdabcdabcdabcd"`.

(a) *(6 points)* First, write a method `retrieveN` that returns the integer at the start of an encoded string. Take the following examples as motivation. Hint: use `indexOf`, `substring`, and `Integer.parseInt`.

```
retrieveN("3[a]2[b]")     => 3
retrieveN("47[abcd]")     => 47
retrieveN("1[bbbbb]3[a]") => 1
```

(b) *(6 points)* Next, write the `cutN` that returns a string without the integer at the start of an encoded string. Hint: use `indexOf` and `substring`.

```
cutN("3[a]2[b]")     => "[a]2[b]
cutN("47[abcd]")     => "[abcd]"
cutN("1[bbbbb]3[a]") => "[bbbbb]3[a]"
```

(c) *(6 points)* Design the *standard recursive* `decode` method, which receives an encoded string and performs a decoding operation. Hint: use s`.repeat(`n`)`, which receives an integer n and operates on a string s, returning a new string with n copies of s.

(d) *(6 points)* Design the `decodeTR` and `decodeTRHelper` methods. The former acts as the driver to the latter; the latter solves the same problem that `decode` does, but it instead uses tail recursion. Remember to include the relevant access modifiers! Hint: use s`.repeat(`n`)`.

(e) *(6 points)* Design the `decodeLoop` method, which solves the problem using either a `while` or `for` loop. Hint: use s`.repeat(`n`)`.

3. (20 points) The *substitution cipher* is a text cipher that encodes an alphabet string A (also called the *plain-text alphabet*) with a key string K (also called the *cipher-text alphabet*). The A string is defined as `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"`, and K is any permutation of A. We can encode a string s using K as a mapping from A. For example, if K is the string `"ZEBRASCDFGHIJKLMNOPQTUVWXY"` and s is `"FLEE AT ONCE. WE ARE DISCOVERED!"`, the result of encoding s produces `"SIAA ZQ LKBA. VA ZOA RFPBLUAOAR!"`

   Design the `subtitutionCipher` method, which receives a plain-text alphabet string A, a cipher-text string K, and a string s to encode, `substitutionCipher` should return a string s' using the aforementioned substitution cipher algorithm. You must follow the "design recipe" laid out in class. That is, you must write the method purpose statement comment, tests, and the implementation.

   The skeleton code is on the next page.

```java
import static Assertions.assertAll;
import static Assertions.assertEquals;

class SubstitutionCipherTester {

  @Test
  void substitutionCipherTest() {







  }
}

import java.util.*; // Import all necessary collections.

class SubstitutionCipher {

  /**
   *
   *
   *
   */
  static String substitutionCipher(String A, String K, String s) {


















  }
}
```

4. (20 points) Oh no! Joshua's cat, Butterscotch, has scratched part of this exam away and we need you to fix the missing code. Fill in the missing code for this quick sort implementation. Note that this is a *functional* implementation of the quick sort, which means that we return a new array rather than sorting the one we provide.

```
import java.util.List;

interface IQuickSort<____________________> {

  List<___> quicksort(List<___> ls);
}


class FunctionalQuickSort implements _______________ {

  @Override
  public List<___> quicksort(List<___> A) {
    if (A.isEmpty()) { return ____; }
    else {
      // Choose a random pivot.
      ___ pivot = new Random().nextInt(_________________________);

      // Sort the left-half.
      List<___> leftHalf = A.stream()
                            .filter(x -> _____________)
                            .collect(Collectors.toList());
      List<___> leftSorted = ____________________;

      // Sort the right-half.
      List<___>  rightHalf = A.stream()
                              .filter(x -> _____________)
                              .collect(Collectors.toList());
      List<___>  rightSorted = _____________________________;

      // Get all elements equal to the pivot.
      List<___>  equal = A.stream()
                          .filter(x -> _____________)
                          .collect(Collectors.toList());

      // Merge the three.
      leftSorted.addAll(equal);
      leftSorted.addAll(rightSorted);
      return leftSorted;
    }
  }
}
```

Scratch work

Scratch work