C212 Final Exam Rubric

1. (60 points) Files on a computer are organized into *directories*, which are just locations for files to exist. Namely, directories can be nested inside of other directories. Therefore, we can categorize a directory as a data definition.

```
A Directory is one of:
 - File
 - new List<Directory>
```

We must then define a File as well, which contains two values: a name and a size (in bytes).

```
A File is a new File(String, Integer)
```

A directory that contains no files will have an empty list of subdirectories. We need a way of labeling that File and Directory are related, so we will define the IContent interface, which contains two methods to denote whether something is a file or a directory.

```
interface IContent {

  /**
   * Determines whether or not the implementing class is a File.
   */
  boolean isFile();

  /**
   * Determines whether or not the implementing class is a Directory.
   */
  boolean isDirectory();
}
```

(a) (10 points) Design the Directory class, which implements IContent, and stores, as an instance variable, a List<IContent>. Then, design the File class, which also implements IContent and stores the two relevant instance variables as described above. Of course, this means you will need to override the isFile and isDirectory methods respectively. **Write your code on the next page.**

```
class Directory implements IContent {
  private List<IContent> loc;
  public Directory() { this.loc = new ArrayList<>(); }

  @Override
  public boolean isFile() { return false; }

  @Override
  public boolean isDirectory() { return true; }
}

class File implements IContent {
  private String filename;
  private int size;
  public File(String fn, int s) {
    this.filename = fn;
    this.size = s;
  }

  @Override
  public boolean isFile() { return true; }

  @Override
  public boolean isDirectory() { return false; }
}
```

(b) (4 points) Design the `void add(IContent c)` method inside `Directory`, which receive a `File/Directory` and adds it to the list of content.

```
void add(IContent c) {
  this.loc.add(c);
}
```

(c) (6 points) Implement the `Comparable` interface for `File` that returns a comparison of the file names. Remember that a class can implement multiple interfaces!

```
class File implements IContent, Comparable<File> {
  // ... other info omitted.

  @Override
  public int compareTo(File f) {
    return this.filename.compareTo(f.filename);
  }
}
```

(d) (8 points) Design the `boolean isPresent(File f)` method inside `Directory`, which determines whether or not a file *f* exists inside the directory instance.

```
boolean isPresent(File f) {
  // Get all directories.
  List<Directory> dirs = this.loc.stream()
                                 .filter(c -> c.isDirectory())
                                 .map(c -> (Directory) c)
                                 .toList();
  // Get all files.
  List<File> files = this.loc.stream()
                             .filter(c -> c.isFile())
                             .map(c -> (File) c)
                             .toList();
  // Check to see if it's in the current directory. If not, recurse.
  if (files.contains(f)) { return true; }
  else {
    for (Directory d : dirs) {
      if (d.isPresent(f)) { return true; }
    }
    return false;
  }
}
```

(e) (8 points) Design the `int countFiles()` method inside `Directory`, which returns the number of files that exist in that directory. It might make sense to write a recursive helper method to solve this problem.

```
int countFiles() {
  return this.countFilesHelper(this);
}

int countFilesHelper(IContent c) {
  if (c.isFile()) { return 1; }
  else {
    int sum = 0;
    for (IContent _c : ((Directory) c).loc) {
      sum += countFilesHelper(_c);
    }
    return sum;
  }
}
```

(f) (8 points) Design the `int countDirectories()` method inside `Directory`, which returns the number of directories that exist in that directory. Do not include the directory itself in this total. It might make sense to write a recursive helper method to solve this problem.

```
int countDirectories() {
  return this.countDirectoriesHelper(this);
}

int countDirectoriesHelper(Directory d) {
  int sum = 0;
  for (IContent c : d.loc) {
    if (c.isDirectory()) {
      sum += 1 + countDirectoriesHelper((Directory) c);
    }
  }
  return sum;
}
```

(g) (6 points) Design the `boolean isEmpty()` method inside `Directory`, which returns whether or not the directory contains any content.

```
boolean isEmpty() {
  return this.loc.isEmpty();
}
```

(h) (10 points) Write coherent tests for your `Directory` and `File` classes. In particular, you should test the following methods: `isPresent`, `countFiles`, `countDirectories`, and `isEmpty`. It might make sense to create a couple of directories outside each test method, then test them inside those methods.

Solution omitted.

2. (30 points) This question has five parts.

Solution.

(a) *(6 points)* First, write the `boolean isVowel(char ch)` method, which returns whether or not the character *ch* is a vowel.

```
isVowel(‘A’) => true
isVowel(‘a’) => true
isVowel(‘X’) => false
isVowel(‘?’) => false
```

```
boolean isVowel(char ch) {
  char u = Character.toUpperCase(ch);
  return u == ‘A’ || u == ‘E’ || u == ‘I’ || u == ‘o’ || u == ‘U’;
}
```

(b) *(6 points)* Next, write the `char swapVowelCasing(char ch)` method, which receives a character and, if it is a vowel, we swap its casing. That is, if it is uppercase, it becomes lowercase, and vice versa. Leave non-vowels alone. The `Character.toUpperCase`, `Character.toLowerCase`, `isUpperCase`, and `isLowerCase` methods will be helpful.

```
swapVowelCasing(‘a’) => ‘A’
swapVowelCasing(‘A’) => ‘a’
swapVowelCasing(‘b’) => ‘b’
swapVowelCasing(‘?’) => ‘?’
```

```
char swapVowelCasing(char ch) {
  if (!isVowel(ch)) { return ch; }
  else {
    if (Character.isUpperCase(ch)) {
      return Character.toLowerCase(ch);
    } else {
      return Character.toUpperCase(ch);
    }
  }
}
```

(c) *(6 points)* Design the *standard recursive* `String swapVowelCasingString(String s)` method, which receives a string and swaps the casing of the vowels thereof.

```
String swapVowelCasingString(String s) {
  if (s.isEmpty()) { return ""; }
  else {
    return swapVowelCasing(s.charAt(0)) +
           swapVowelCasingString(s.substring(1));
  }
}
```

(d) *(6 points)* Design the `String swapVowelCasingStringTR(String s)` as well as the `String swapVowelCasingStringTRHelper(...)` methods. The former acts as the driver to the latter; the latter solves the same problem that `swapVowelCasingString` does, but it instead uses tail recursion. Remember to include the relevant access modifiers!

```
String swapVowelCasingStringTR(String s) {
  return swapVowelCasingStringTRHelper(s, "");
}

private String swapVowelCasingStringTRHelper(String s, String acc) {
  if (s.isEmpty()) { return acc; }
  else {
    return swapVowelCasingStringTRHelper(s.substring(1),
                                         acc +
                                         swapVowelCasing(s.charAt(0)));
  }
}
```

(e) *(6 points)* Design the `String swapVowelCasingStringLoop(String s)` method, which solves the problem using either a `while` or `for` loop.

```
String swapVowelCasingStringLoop(String s) {
  String acc = "";
  while (!s.isEmpty()) {
    acc = acc + swapVowelCasing(s.charAt(0));
    s = s.substring(1);
  }
  return acc;
}
```

3. (20 points) **Solution.**

We consider a *key string* to be the string obtained after alphabetizing the letters of a string. For example, the string "deloop" is a key string of the strings "poodle" and "looped". Write the static HashMap<String, List<String>> keyStringGroups(List<String> ls) method, which maps all key strings to the strings in *ls* using the above criteria. We provide an example below. You **cannot** use this example in your tests.

```
ls = ["ant", "introduces", "poodle", "tan", "looped", "discounter", "nastier",
      "polled", "retains", "retinas", "reductions"]

keyStringGroups(ls) => {{"ant", ["tan", "ant"]},
                        {"deloop", ["poodle", "looped"]},
                        {"dellop", ["polled"]},
                        {"cdeinorsu", ["discounter", "introduces", "reductions"]},
                        {"aeinrst", ["retains", "retinas", "nastier"]}}
```

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

class KeyStringGroup {

  static HashMap<String, List<String>> keyStringGroups(List<String> ls) {
    HashMap<String, List<String>> ksg = new HashMap<>();

    // For all strings s, create the corresponding "key string",
    // add it to the map. Then, for all strings, find its bucket.
    for (String s : ls) {
      char[] chs = s.toCharArray();
      Arrays.sort(chs);
      ksg.put(new String(chs), new ArrayList<>());
    }

    for (String s : ls) {
      char[] chs = s.toCharArray();
      Arrays.sort(chs);
      ksg.get(chs).add(s);
    }

    return ksg;
  }
}
```

4. (20 points) Two strings s_1 and s_2 are isomorphic if we can create a mapping from s_1 from s_2. For example, the strings `"DCBA"` and `"ZYXW"` are isomorphic because we can map D to Z, C to Y, and so forth. Another example is `"ABACAB"` and `"XYXZXY"` for similar reasons. A non-example is `"PROXY"` and `"ALPHA"`, because once we map `"A"` to `"P"`, we cannot create a map between `"A"` and `"Y"`. Write the `isIsomorphic` method, which determines whether or not two strings are isomorphic. Follow the design recipe from class. That is, write the purpose statement, followed by a sequence of examples, then the definition. **The skeleton code is on the next page.**

Solution. *This is a pretty difficult problem and was intended to be on the actual exam. After completing it, I'm glad that we changed our minds...*

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

class IsomorphicString {

  static boolean isIsomorphic(String s, String t) {
    // Create two maps for each mapping.
    HashMap<Character, Character> sM1 = new HashMap<>();
    HashMap<Character, Character> sM2 = new HashMap<>();

    // They can't be isomorphic if they are of different lengths.
    if (s.length() != t.length()) { return false; }
    else {
      for (int i = 0; i < s.length(); i++) {
        char c1 = s.charAt(i);
        char c2 = t.charAt(i);
        // If the map doesn't contain the keys, add the mapping.
        if (!sM1.containsKey(c1) && !sM2.containsKey(c2)) {
          sM1.put(c1, c2);
          sM2.put(c2, c1);
        }
        // If exactly one of them doesn't contain the mapping, it
        // cannot be isomorphic.
        else if (!sM1.containsKey(c1) || !sM2.containsKey(c2)) {
          return false;
        }
        // If we try to map to an already mapped value that
        // does not match, it cannot be isomorphic.
        else if (sM1.get(c1) != c2 || sM2.get(c2) != c1) {
          return false;
        }
      }
      return true;
    }
  }
}
```

5. (20 points) Oh no! Joshua's cat, Nebraska, has scratched part of this exam away and we need you to fix the missing code. Fill in the missing code for this merge sort implementation. Note that this is a *in-place* implementation of the merge sort, meaning that we modify the list in-place, rather than returning a new one. Only the `mergeSort` and `merge` methods should be filled in.

   Solution.

```
import java.util.List;

interface IMergeSort<T extends Comparable<T>> {

  List<T> mergeSort(List<T> ls);
}

class InPlaceMergeSort<T extends Comparable<T>> implements IMergeSort<T> {

  private void mergeSortHelper(List<T> ls, int low, int high) {
    if (low < high) {
      int mid = (low + (high - low)) / 2;
      mergeSortHelper(ls, low, mid);
      mergeSortHelper(ls, mid + 1, high);
      merge(ls, low, mid, high);
    }
  }

  private void merge(List<T> ls, int low, int mid, int high) {
    List<T> left = new ArrayList<>();
    List<T> right = new ArrayList<>();

    for (int i = low; i <= mid; i++) { left.add(ls.get(i)); }
    for (int j = mid + 1; j <= high; j++) { right.add(ls.get(j)); }

    int mergedIdx = low;
    int i = 0;
    int j = 0;

    while (i < left.size() && j < right.size()) {
      if (left.get(i).compareTo(right.get(j)) < 0) {
        ls.set(mergedIdx++, left.get(i++));
      } else {
        ls.set(mergedIdx++, right.get(j++)); }
    }

    while (i < left.size()) { ls.set(mergedIdx++, left.get(i++)); }
    while (j < right.size()) { ls.set(mergedIdx++, right.get(j++)); }
  }
}
```

Scratch work

Scratch work