CSE 331 Fall 2017 *Practice* Final Exam

Name ___________________Solution________________________

The exam is closed book and closed electronics. One page of notes is allowed.

Please **wait to turn the page** until everyone is told to begin.

Score: __________________ / ?

1. _______________ / ?
2. _______________ / ?
3. _______________ / ?
4. _______________ / ?
5. _______________ / ?
6. _______________ / ?

In problems 1–5, we write an implementation of the following interface:

```
/** Synonym for Iterator<List<Song>>. */
public interface PlayListIterator extends Iterator<List<Song>> {}
```

A `PlayList` is simply a list of songs.

Each `Song` has the following interface:

```
/** Information about a particular piece of music. */
public class Song {

  /** Returns the name of the song. */
  public String getName() { .. }

  /** Returns the name of the artist performing the song. */
  public String getArtist() { .. }

  /** Returns the name of the album containing the song. */
  public String getAlbum() { .. }

}
```

Recall that the interface `Iterator<T>` includes the following methods[1]:

```
/** Produces a stream of objects of type T. */
public iterator Iterator<T> {

  /** Determines whether there is another object to return. */
  public boolean hasNext();

  /** Returns the next object in the stream or throws
    * NoSuchElementException if there are no more remaining. */
  public T next() throws NoSuchElementException;

}
```

Here, the term "stream" means that the objects are produced one-at-a-time in order and then forgotten by the iterator. There is no way to go back to earlier elements.

[1] `Iterator` also contains a `remove` method, but we will ignore that here.

Problem 1 (ADT)

In this problem, you will design an ADT, `GenrePlayListIterator`, that implements `PlayListIterator` by looking through the user's library of songs.

You can assume the existence of the following method in your class:

```
/**
 * Find all songs in the first genre countered that is not skipped.
 * @param library List of songs in the user's library.
 * @param start Ignore songs in the library with indexes below this.
 * @param albumGenres Maps album names to their genres.
 * @param skipGenres Ignore all songs with genres in this list.
 * @param songs Output variable for the list of songs found.
 * @requires library, albumGenres, & skipGenres are non-null and
 *     0 <= start < library.size()
 * @modifies songs if non-null
 * @effects Adds all the songs in library[start..] whose genre is
 *     the same as that of the first non-skipped genre encountered
 * @returns Returns the index of the first song encountered that has
 *     a non-skipped genre or -1 if none is encountered
 */
private static int findSongsByGenre(
    List<Song> library, int start, Map<String, String> albumGenres,
    Set<String> skipGenres, List<Song> songs) { .. }
```

Your ADT will have the following fields.

```
// RI: parts (1-3) below:
// (1) ??
// (2) genresSeen contains all genres returned previously by next()
// (3) start is the smallest value in [0..library.size()] that is
//     strictly larger than the first index at which each genre in
//     genresSeen appears in libary
private final List<Song> library;
private final Map<String, String> albumGenres;
private final Set<String> genresSeen;
private int start;
```

a. Fill in the obvious parts of the RI that are missing in part (1):

library != null, albumGenres != null, genresSeen != null

b. Fill in the blank to complete the abstraction function for your ADT:

AF(this) = stream of playlists, each containing all songs with a particular genre,
with one list for each genre in library not included in __genresSeen__,
returned in the order they first appear in library

c. Fill in the code for the constructor:

```
/**
 * Creates an iterator that produces playlists for each album
 * genre found in the given library.
 * @param library List of songs in the user's library.
 * @param albumGenres Maps albums to their genre
 * @requires library & albumGenres are non-null
 */
public GenrePlayListIterator(
    List<Song> library, Map<String, String> albumGenres) {
  this.library = library;
  this.albumGenres = albumGenres;
  this.genresSeen = new HashSet<String>();
  this.start = 0;

}
```

d. Fill in the implementation of `hasNext` (Hint: use `findSongsByGenre`):

```
@Override public boolean hasNext() {

  return (start < library.size()) && findSongsByGenre(
      library, start, albumGenres, genresSeen, null) >= 0;


}
```

e. Fill in the rest of the implementation of `next` (Hint: use `findSongsByGenre`):

```
@Override public List<Song> next() throws NoSuchElementException {
  if (!hasNext()) {
    throw new NoSuchElementException();
  } else {
    List<Songs> = new ArrayList<Songs>();
    int nextIndex = findSongsByGenre(
        library, start, albumGenres, genresSeen, songs);
    this.start = (nextIndex >= 0) ? nextIndex+1 : start;
    this.genresSeen.add(albumGenres.get(songs.get(0).getAlbum()));
    return songs;
  }
}
```

f. If it is not obvious, explain why your implementation leaves the object in a state that satisfies each parts of the RI:

If nextIndex >= 0, then nextIndex+1 is one larger than the first song in enery seen genre (since start was > all but the one just returned). Otherwise, we've seen them all, so start was already the smallest value in that range.

g. Explain what will happen in your implementation for albums whose genre is not included in the map `albumGenres` (Note: it should not crash!):

   genresSeen will end up containing null, and all songs that are not included in the map albumGenres will be returned in the playlist for genre null.

h. **Briefly** describe how you would check RI part (3) in your `checkRep` (you do not need to write out the code in full detail):

   For each genre in genresSeen, find the first index that it appears. Put those in a list. If the list is empty, then start = 0. Otherwise, check that we have start = max(first index + 1) over the first indexes in the list.

Problem 3 (Reasoning)

Fill in the implementation of `findSongsByGenre` below. Your implementation should use only the loops shown and should be correct using the **invariants given below**.

(The javadoc for this method was given on a previous page.)

```
private static int findSongsByGenre(
    List<Song> library, int start, Map<String, String> albumGenres
    Set<String> skipGenres, List<Song> songs) {

  int index = 0;


  {{ Inv: start <= index <= library.size() and
         every album in library[start..index-1] is in skipGenres }}
  while (index < library.size() && skipGenres.contains(
            albumGenres.get(library.get(index).getAlbum()))) {
    index++;
  }

  if (index == library.size())
    return -1;


  String songGenre = albumGenres.get(library.get(index).getAlbum());
  {{ songGenre is the album of library[index] and is not in skipGenres }}

  if (songs == null) return index;
  songs.add(library.get(index));
  start = index;

  {{ Inv: songs is non-null and contains every song in
         library[start..index] whose genre is songGenre }}
  while (index+1 < library.size()) {
    String genre = albumGenres.get(library.get(index+1).getAlbum());
    if (songGenre == genre || genre != null && genre.equals(songGenre))
      songs.add(library.get(index+1));

    index++;


  }

  return start;
}
```

Problem 2 (Testing)

In this problem, you will write test cases for your ADT. Each test case describes the library input to the constructor and the output produced by calling `next` repeatedly until a `NoSuchElementException` is thrown.

You can write each individual song as a triple of the form (name, artist, genre). Instead of describing how albums are mapped to genres, we will simply list the genres.

Make sure that each input tests a distinct subdomain in terms of either the expected or actual behavior of your constructor, `next` , or `findSongsByGenre`. If it is not clear why a test case is in a distinct subdomain, **write an explanation below the test**.

a. Input: library = []

   Output: [[]]

b. Input: library = [("1", "1", "A"), ("2", "2", "B")]

   Output: [[("1", "1", "A")], [("2", "2", "B")]]

   library is not empty

c. Input: library = [("1", "1", "A"), ("2", "2", "A")]

   Output: [[("1", "1", "A"), ("2", "2", "A")]]

   multiple songs in the same genre

d. Input: library =[("1", "1", "A"), ("2", "2", "B"), ("3", "3", "A")]

   Output: [[("1", "1", "A"), ("3", "3", "A")], [("2", "2", "B")]

   mulltiple songs in the same genre that are non-adjacent

e. What part of the `PlayListIteratoIterator` API is **not tested at all** by these tests when implemented as described in the opening paragraph?

   `hasNext` is never tested

Problem 5 (Reasoning II)

In this problem, you will implement a method for computing the maximum sum of any subarray of a given array. More specifically, it finds the (i,j) that maximizes the sum vals[i] + ... + vals[j], where vals is a given array of integers.

```
/**
 * Returns the indexes of the subarray with maximum sum.
 * @param vals Array of values
 * @returns {a, b} such that sum of vals[a..b] is the maximum of
 *      vals[i..j] over all choices of i and j
 */
public static int[] maxSubarraySum(int[] vals) {

  int[] maxSum = int new[vals.length+1];
  int[] maxStart = int new[vals.length+1];
  int index = 0;
  int maxIndex = ______________;

  // Inv: parts (1-3) below:
  // (1) for j=0..index, maxSum[j] =
  //       maximum sum of vals[i..j-1] over all choices of i
  // (2) for j=0..index, maxStart[j] satisfies
  //       maxSum[j] = sum of vals[maxStart[j]..j-1]
  // (3) maxSum[maxIndex] is the maximum of maxSum[0..index]

  while (_________________________) {
```

a. What should we set `maxIndex` to so that Inv is initially true?

0

b. How do we choose the loop condition so that we end with `maxSum[maxIndex]` being the maximum subarray sum? (Be careful here!)

`index+1 <= vals.length`

c. Write a return statement, to go **after the loop**, so that it satisfies the specification:

`return new int[] { maxStart[maxIndex], maxIndex-1 };`

It remains only to write the body of the loop.

d. We will increase `index` by 1 on each iteration. What **additional** claims are made in Inv when `index` changes to `index+1`? Write them for each of the parts:

   (1) maxSum[index+1] = max(sum of vals[i..index])

   (2) maxSum[index+1] = sum(vals[maxStart[index+1]..index])

   (3) maxSum is the maximum of maxSum[0..index+1]

Let's consider how to compute maxSum[index+1] = max(sum of vals[i..index]).

The possible values of i to consider here are i = 0..index+1. When i >= index+1, we have an empty subarray (vals[index+1..index]), whose sum is 0.

For any i <= index, we have a non-empty subarray because it includes at least the element vals[index]. This means we can write:

    sum of vals[i..index] = (sum of vals[i..index-1]) + vals[index]

Now, consider what happens if we take max(sum of vals[i..index]) over all i <= index. Since vals[index] is the same for all i, this is max(sum of vals[i..index-1]) + vals[index].

e. Write a short Java expression to compute max(sum of vals[i..index-1]) + vals[index]. (Hint: use the RI.)

```
maxSum[index] + vals[index]
```

The value of max(sum of vals[i..index]) is either the sum for an i >= index+1 or the sum for an i <= index.

f. Write a short Java expression that determines whether the maximum is achieved by some i <= index rather than by i >= index+1.

```
maxSum[index] + vals[index] > 0
```

g. Fill in code to ensure that RI parts (1-2) are still satisfied when index is incremented in the case that max(sum of vals[i..index]) is achieved by some i <= index:

```
maxSum[index+1] = maxSum[index] + vals[index];
maxStart[index+1] = maxStart[index];
```

h. Fill in code to ensure that RI parts (1-2) are still satisfied when index is incremented in the case that max(sum of vals[i..index]) is achieved by some i >= index+1:

```
maxSum[index+1] = 0;
maxStart[index+1] = index+1; // maxStart[index+1]..index] is empty
```

i. Fill in code to ensure that RI part (3) is satisfied when index is incremented. (This should work for either of the two cases considered above, and it should not require any additional loops!)

```
if (maxSum[index+1] > maxSum[maxIndex])
  maxIndex = index+1;
```

This completes the implementation of `maxSubarraySum`.

Problem 6 (Miscellaneous)

a. The Song class has the following constructor:

```
Song(String name, String artist, String album) { .. }
```

What is worrisome about this?

The client could easily confuse the order of the arguments.

b. In what way is `findSongsByGenre` an example of bad method design? How could the author have designed it to fix that problem?

It does two things: find the index of the next unseen genre and collect a list of the songs in that genre. (In the method, they are even done one after the other.) This should really be two separate methods.