The declaration for the `Flight` class is shown below. It has methods to access the departure time and the arrival time of a flight. You may assume that the departure time of a flight is earlier than its arrival time.

```
public class Flight
{
   /** @return time at which the flight departs
    */
   public Time getDepartureTime()
   { /* implementation not shown */ }

   /** @return time at which the flight arrives
    */
   public Time getArrivalTime()
   { /* implementation not shown */ }

  // There may be instance variables, constructors, and methods that are not shown.
}
```

GO ON TO THE NEXT PAGE.

A trip consists of a sequence of flights and is represented by the `Trip` class. The `Trip` class contains an `ArrayList` of `Flight` objects that are stored in chronological order. You may assume that for each flight after the first flight in the list, the departure time of the flight is later than the arrival time of the preceding flight in the list. A partial declaration of the `Trip` class is shown below. You will write two methods for the `Trip` class.

```
public class Trip
{
  private ArrayList<Flight> flights;
     // stores the flights (if any) in chronological order

  /** @return the number of minutes from the departure of the first flight to the arrival
   *          of the last flight if there are one or more flights in the trip;
   *          0, if there are no flights in the trip
   */
  public int getDuration()
  {   /* to be implemented in part (a) */   }

  /** Precondition: the departure time for each flight is later than the arrival time of its
   *                 preceding flight
   *   @return the smallest number of minutes between the arrival of a flight and the departure
   *           of the flight immediately after it, if there are two or more flights in the trip;
   *           -1, if there are fewer than two flights in the trip
   */
  public int getShortestLayover()
  {   /* to be implemented in part (b) */   }

  // There may be instance variables, constructors, and methods that are not shown.
}
```

The class `StringCoder` provides methods to encode and decode words using a given master string. When encoding, there may be multiple matching string parts of the master string. The helper method `findPart` is provided to choose a string part within the master string that matches the beginning of a given string.

```
public class StringCoder
{
  private String masterString;

  /** @param master  the master string for the StringCoder
   *            Precondition: the master string contains all the letters of the alphabet
   */
  public StringCoder(String master)
  {  masterString = master;  }


  /** @param parts  an ArrayList of string parts that are valid in the master string
   *            Precondition: parts.size() > 0
   *  @return  the string obtained by concatenating the parts of the master string
   */
  public String decodeString(ArrayList<StringPart> parts)
  {   /*  to be implemented in part (a)  */   }


  /** @param str  the string to encode using the master string
   *            Precondition: all of the characters in str appear in the master string;
   *                          str.length() > 0
   *  @return  a string part in the master string that matches the beginning of str.
   *            The returned string part has length at least 1.
   */
  private StringPart findPart(String str)
  {   /*  implementation not shown  */   }


  /** @param word  the string to be encoded
   *            Precondition: all of the characters in word appear in the master string;
   *                          word.length() > 0
   *  @return  an ArrayList of string parts of the master string that can be combined
   *            to create word
   */
  public ArrayList<StringPart> encodeString(String word)
  {   /*  to be implemented in part (b)  */   }

  //  There may be instance variables, constructors, and methods that are not shown.
}
```

GO ON TO THE NEXT PAGE.

(a) Write the `StringCoder` method `decodeString`. This method retrieves the substrings in the master string represented by each of the `StringPart` objects in `parts`, concatenates them in the order in which they appear in `parts`, and returns the result.

Complete method `decodeString` below.

```
/** @param parts an ArrayList of string parts that are valid in the master string
 *          Precondition: parts.size() > 0
 *  @return the string obtained by concatenating the parts of the master string
 */
public String decodeString(ArrayList<StringPart> parts)
```

GO ON TO THE NEXT PAGE.

(b) Write the `StringCoder` method `encodeString`. A string is encoded by determining the substrings in the master string that can be combined to generate the given string. The encoding starts with a string part that matches the beginning of the word, followed by a string part that matches the beginning of the rest of the word, and so on. The string parts are returned in an array list in the order in which they appear in `word`.

The helper method `findPart` must be used to choose matching string parts in the master string.

Complete method `encodeString` below.

```
/** @param word the string to be encoded
 *          Precondition: all of the characters in word appear in the master string;
 *                        word.length() > 0
 *  @return an ArrayList of string parts of the master string that can be combined
 *           to create word
 */
public ArrayList<StringPart> encodeString(String word)
```

GO ON TO THE NEXT PAGE.

3. This question involves reasoning about the code from the GridWorld case study. A copy of the code is provided as part of this exam.

An opossum is an animal whose defense is to pretend to be dead. The `OpossumCritter` class, shown below, will be used to represent the opossum in the grid. An `OpossumCritter` classifies its neighbors as friends, foes, or neither. It is possible that a neighbor is neither a friend nor a foe; however, no neighbor is both a friend and a foe. If the `OpossumCritter` has more foes than friends surrounding it, it will simulate playing dead by changing its color to black and remaining in the same location. Otherwise, it will behave like a `Critter`. If the `OpossumCritter` plays dead for three consecutive steps, it is removed from the grid.

You will implement two of the methods in the following `OpossumCritter` class.

```
public class OpossumCritter extends Critter
{
  private int numStepsDead;

  public OpossumCritter()
  {
    numStepsDead = 0;
    setColor(Color.ORANGE);
  }

  /** Whenever actors contains more foes than friends, this OpossumCritter plays dead.
   *  Postcondition: (1) The state of all actors in the grid other than this critter and the
   *  elements of actors is unchanged. (2) The location of this critter is unchanged.
   *  @param actors a group of actors to be processed
   */
  public void processActors(ArrayList<Actor> actors)
  { /* to be implemented in part (a) */ }


  /** Selects the location for the next move.
   *  Postcondition: (1) The returned location is an element of locs, this critter's current location,
   *  or null. (2) The state of all actors is unchanged.
   *  @param locs the possible locations for the next move
   *  @return the location that was selected for the next move, or null to indicate
   *          that this OpossumCritter should be removed from the grid.
   */
  public Location selectMoveLocation(ArrayList<Location> locs)
  { /* to be implemented in part (b) */ }
```

GO ON TO THE NEXT PAGE.