Section Solutions 5

Problem One: Selection Sort

Here is one possible solution:

```
import acm.program.*;
import java.util.*;
public class SelectionSort extends ConsoleProgram {
  /* The sentinel value that marks the end of the input. */
  private static final int SENTINEL = -1;

  /**
   * Read a list of values from the user, then print them in sorted
   * order.
   */
  public void run() {
    printSorted(readValues());
  }

  /**
   * Reads a list of integers from the user, stopping when the user enters
   * the sentinel, and returns all the values that were read.
   *
   * @return The list of values entered by the user.
   */
  private ArrayList<Integer> readValues() {
    ArrayList<Integer> result = new ArrayList<Integer>();

    while (true) {
      /* Read another value from the user, stopping if the user enters
       * the sentinel.
       */
      int value = readInt("Enter value or " + SENTINEL + " when done: ");
      if (value == SENTINEL) return result;

      result.add(value);
    }
  }

  /**
   * Given a list of numbers, prints that list in sorted order. This will
   * destructively modify the input list.
   *
   * @param toPrint The list to print, which is emptied as a result.
   */
  private void printSorted(ArrayList<Integer> toPrint) {
    /* While there are still values in the array, rmove and print the
     * smallest.
     */
    while (!toPrint.isEmpty()) {
      println(removeSmallest(toPrint));
    }
  }
  /**
   * Given a nonempty list, removes and returns the smallest number from that
   * list.
   *
   * @param list The list of values, which will be modified by having its
   *        smallest value removed.
   * @return The smallest value from that list.
   */
  private int removeSmallest(ArrayList<Integer> list) {
    /* Iterate over the list and keep track of the index of the smallest
     * element in the array.
     */
    int smallestIndex = 0;
    for (int i = 1; i < list.size(); i++) {
      /* If the current element is smaller than the smallest known
       * element, update the index of the smallest element.
       */
      if (list.get(i) < list.get(smallestIndex)) {
        smallestIndex = i;
      }
    }

    /* Remove and return the smallest element. */
    int result = list.get(smallestIndex);
    list.remove(smallestIndex);
    return result;
  }
}
```

Problem Two: Method Testing

```
/**
 * Given a string representing a single word, estimates the number of syllables
 * in that word by counting the number of groups of vowels, except for isolated
 * e's at the end of the word. (Y counts as a vowel). All words
 *
 * @param word The word in question
 * @return An estimate of the number of syllables as described above.
 */
private int syllablesInWord(String word)
```

This method needs to look at groups of vowels. As some basic tests, it wouldn't hurt to look at different strings with different groups of vowels in them (groups of sizes one, two, or three, for example). We should check for edge cases where those groups are at the very beginning or end of a word. That means we might want to test strings like

- Beau
- Auburn
- Monkey

The method also needs to special-case singleton e's at the end of the word. We should make sure that it correctly handles isolated e's, but doesn't incorrectly drop non-isolated e's. This gives tests like

- Manatee
- Eve
- Believe

We also want to ensure that the method always returns at least one syllable when it otherwise would report nothing. We could use tests like

- Me
- Be

It doesn't hurt to check that every vowel is correctly counted as a vowel. We can construct a contrived input like this one to check for that:

- lalelilolulyl

Then we need to make sure that the method is case-insensitive, so it can't hurt to try upper-case and lower-case versions of the above strings.

What other tests might you run?

```
/**
 * Given two numeric strings (strings consisting purely of digits) representing two
 * numbers n1 and n2, returns a numeric string representing their sum. The input
 * strings don't have to be the same length, but each will represent a nonnegative
 * integer.
 *
 * @param n1 The string encoding of the first number
 * @param n2 The string encoding of the second number.
 * @return A string encoding of their sum.
 */
private String addNumericStrings(String n1, String n2)
```

We can start with some basic checks to see if the strings add correctly when we have inputs of the same length and without any carrying involved. We'll do both long and short strings to make sure that they're handled correctly:

- 0 + 1
- 1111111111111111 + 2222222222222222
- 8888888 + 1111111

Next, let's introduce carrying, but where the carry doesn't propagate to the front of the string.

- 19 + 11
- 18888 + 11112

Now, carrying where it does propagate:

- 1 + 9
- 5 + 5
- 55 + 45
- 88888 + 11112

Next, we'll check strings of different length without carrying:

- 18 + 1
- 8888 + 1000

Now, strings of different lengths where some carrying does occur, but doesn't propagate all the way to the front:

- 188 + 2
- 1237 + 163

Now, strings of different lengths where some carrying does occur and does propagate to the front.

- 99999999999999999999999 + 1

What other tests might you run?

```
/**
 * Given as input a string of three letters and a list of strings, returns all
 * words in English that match that word according to the rules of the “license
 * plate game” (that is, all words that contain all of the letters in the string
 * 'letters' in the order in which they appear).
 *
 * @param letters A string of three letters.
 * @param allWords A list of all the strings to test.
 * @return A list of all the English words matching the given letter patter.
 */
private ArrayList<String> allMatchesFor(String letters, ArrayList<String> allWords)
```

One major simplification you can make to ease testing is to not use the full dictionary and to instead craft specific word lists that exercise specific concepts. We'll begin by using lists of just one word so that we can test whether the letters match against it.

We should start by making sure that any three letters match itself:

- "the" and ["the"] (true)

Also, that three letters don't match a permutation of themselves:

- "the" and ["het"]

Now, let's try some samples where the three letters appear consecutively in a longer string, making sure to position the string in different spots to check for edge cases:

- "the" and ["other"] (true)
- "the" and ["theater"] (true)
- "the" and ["lithe"] (true)

Next, let's start introducing cases where all three letters are present but spaced apart. We'll try different letter spacings to see how they work:

- "ioa" and ["isogram"] (true)
- "soa" and ["isogram"] (true)
- "sam" and ["isogram"] (true)

It's good to check for case sensitivity:

- "the" and ["THE"]
- "THE" and ["the"]
- "the" and ["tHe"]

Now, let's see if duplicate letters work correctly:

- "aaa" and ["a"] (false)
- "aaa" and ["banana"] (true)
- "aan" and ["anna"] (false)
- "aan"and ["banana"] (true)

Now, we need some tests to make sure this code works across longer word lists. What sorts of test might you design for that?

```
/**
 * Tokenizes the input CSV file line and returns all the fields in that line.
 *
 * @param line A line from a CSV file.
 * @return A list of all the tokens in that line, with any external quotation marks
 *         removed.
 */
private ArrayList<String> fieldsIn(String line)
```

We need to make sure this works on simple inputs where there are no spaces anywhere:

- `man,plan,canal,panama`

Next, let's see if it respects spaces. This should tokenize, but have spaces before the second and third fields:

- `why, oh, why?`

Fields might be empty. This should have an empty field at the end:

- `an,extra,comma,`

Everything here should be empty:

- `,,,,,`

How about a line with just one field:

- `solipsism`

Now, let's introduce quotes, but with no commas. This should come back with no quotes:

- `"hello","goodbye"`

Quotes just in some places but not others:

- `"hello",goodbye`
- `hello,"goodbye"`

Just one field, and it's quoted:

- `"hello"`

Now, quotes with commas in them:

- `"tricky,tricky","tricky,tricky"`
- `"tricky,tricky",tricky,tricky`
- `tricky,"tricky,tricky",tricky`
- `"tricky,tricky,tricky,tricky"`

What other tests might you run?