CS106A Practice Midterm 2 Solutions

Problem 1: Isograms (10 Points)

There are many possible solutions to this problem. Here are six of them:

```
private boolean isIsogram(String word) {
  word = word.toLowerCase();
  String used = "";

  for (int i = 0; i < word.length(); i++) {
    char ch = word.charAt(i);

    if (used.indexOf(ch) != -1)
      return false;
    used += ch;
  }

  return true;
}
```

```
private boolean isIsogram(String word) {
  word = word.toLowerCase();
  for (int i = 0; i < word.length(); i++) {
    char ch = word.charAt(i);

    if (word.indexOf(ch, i + 1) != -1)
      return false;
  }

  return true;
}
```

```
private boolean isIsogram(String word) {
  word = word.toLowerCase();
  for (int i = 0; i < word.length(); i++) {
    char ch = word.charAt(i);

    for (int j = i + 1; j < word.length();
         j++) {
      if (word.charAt(j) == ch)
        return false;
    }
  }

  return true;
}
```

```
private boolean isIsogram(String word) {
  word = word.toLowerCase();
  HashMap<Character, Boolean> used =
    new HashMap<Character, Boolean>();

  for (int i = 0; i < word.length(); i++) {
    char ch = word.charAt(i);

    if (used.containsKey(ch)) return false;
    used.put(ch, true);
  }

  return true;
}
```

```
private boolean isIsogram(String word) {
  word = word.toLowerCase();
  for (char ch = 'a'; ch <= 'z'; ch++) {
    boolean found = false;
    for (int i = 0; i < word.length(); i++) {
      if (word.charAt(i) == ch) {
        if (found) return false;
        found = true;
      }
    }
  }
  return true;
}
```

```
private boolean isIsogram(String word) {
  word = word.toLowerCase();
  boolean[] used = new boolean[26];

  for (int i = 0; i < word.length(); i++) {
     char ch = word.charAt(i);

     if (used[ch - 'a']) return false;
     used[ch - 'a'] = true;
  }

  return true;
}
```

In case you're curious, the longest isogram in English is the word “subdermatoglyphic,” which means “pertaining to features of the skin determined by the layer of skin just below the surface.” Considering how long that description is, it's no wonder there's a word for it.

You can extend the idea of isograms to sentences that don't repeat any letters. Interestingly, there are several English sentences that use each letter exactly once. They're all pretty weird and either borrow from other languages or use acronyms that were later accepted as English words. For example:

Veldt jynx grimps waqf zho buck

As I'm typing this, my word processor is marking each of the above words (save for “buck”) as spelled incorrectly, though I promise they're real words. Honest.

Problem Two: Announcing Election Results (10 Points)

Here are two different solutions. Both solutions work by counting up how many people voted for each candidate, but use different approaches.

```
private String electionWinner(String[] votes) {
    for (String person: votes) {
       if (isWinner(person, votes)) return person;
    }
    return null;
}

private boolean isWinner(String person, String[] votes) {
    int total = 0;
    for (String otherPerson: votes) {
        if (person.toLowerCase().equals(otherPerson.toLowerCase()) total++;
    }
    return total > votes.length / 2;
}
```

```
private String electionWinner(String[] votes) {
    HashMap<String, Integer> counts = new HashMap<String, Integer>();
    for (String person: votes) {
        person = person.toLowerCase();
        if (!counts.containsKey(person)) {
            counts.put(person, 0);
        }
        counts.put(person, counts.get(person) + 1);
        if (counts.get(person) > votes.length / 2) return person;
    }
    return null;
}
```

Why we asked this question: This question was designed to test whether you were able to correctly count up the number of occurrences of various values in a list. There are many ways to do this, as shown above. We also wanted to test whether you remembered how to compare strings (using `.equals` versus `==`) and how to treat strings case-insensitively.

Common mistakes: Overall, the class did well on this problem. Some solutions using `HashMap` forgot to add an initial key/value pair to the map for every candidate, which would cause the program to work incorrectly when it saw the very first instance of each name. Many solutions attempted to be case-insensitive but had a mistake while doing so (either forgetting that `.toLowerCase()` returns a lower-case version of the string rather than converting the string to lower case, or by using `.toLowerCase()` inconsistently). We also saw many solutions that found someone with a *plurality* of the votes rather than a *majority* of the votes, which is an interesting exercise but wasn't quite what we were asking for.

Problem Three: Short Answer (10 Points)

(i) Comparing Data Structures, Part One (4 Points)

Here are some advantages of `String` over **`char`**`[]`:

- `String` is immutable, so passing a `String` into a method guarantees the text comes back unchanged after the method returns. **`char`**`[]` doesn't have this property.
- It is significantly easier to concatenate, split, search, etc. a `String` than a **`char`**`[]`.
- It is possible to create a `String` to hold some text without precomputing the size of that text, which must be done for **`char`**`[]`.

The main advantage of **`char`**`[]` over `String` is that **`char`**`[]` is mutable, so methods that need to modify text can be easier to write using **`char`**`[]` than `String`.

(ii) Comparing Data Structures, Part Two (3 Points)

Here are some advantages of `ArrayList` over `HashMap`:

- Iterating over an `ArrayList` hands back elements in order; iterating over a `HashMap` does not.
- `ArrayList` is bounds-checked, so reading off the end of an `ArrayList` triggers an error at the spot where the read was attempted. Reading off a `HashMap` will just return `null`, which might cause errors later on.
- Removing an element from an `ArrayList` shuffles all elements after it down one step. This is not the case in a `HashMap`.
- The `size()` of an `ArrayList` is always one past the last entry. In a `HashMap`, keys aren't necessarily contiguous, so this isn't the case.
- `ArrayList` supports native operations on sequences like `indexOf`, etc.

Here are some advantages of `HashMap` over `ArrayList`:

- The keys in a `HashMap` don't have to be consecutive, so if you only need to store a few elements from a sequence, `HashMap` might be easier to use.
- `HashMap`s can have negative indices, while `ArrayList`s cannot.

(iii) An iOS Vulnerability **(3 Points)**

```
 1: private boolean hasSmallDivisor(int input) {
 2:    boolean result = false;
 3:
 4:    if (input % 2 == 0)
 5:        result = true;
 6:
 7:    if (input % 3 == 0)
 8:        result = true;
 9:        result = true;
10:
11:    if (input % 5 == 0)
12:        result = true;
13:
14:    return result;
15: }
```

The issue here is that line 9 is not controlled by the if statement it appears to be nested inside of. The above code is equivalent to this code:

```
private boolean hasSmallDivisor(int input) {
    boolean result = false;

    if (input % 2 == 0) result = true;
    if (input % 3 == 0) result = true;
    result = true;
    if (input % 5 == 0) result = true;
    return result;
}
```

Here, it's more obvious that result is always set to true, regardless of what the input is.

Why we asked this question: Parts (i) and (ii) of this problem were designed to get you thinking about tradeoffs in representations. You've seen different ways of representing the same objects, and those options have different benefits and drawbacks. We wanted to see if you'd thought through those tradeoffs.

I included part (iii) here so that you could (hopefully) see how the skills you've picked up in CS106A can better help you understand real-world software bugs. Most programmer errors aren't terribly complicated, and in this case the real security vulnerability was a fundamental mixup in concepts related to CS106A.

Common mistakes: For part (i), many people mentioned that `String` is an abstract data type while `char[]` is not but did not elaborate on why this was an advantage or disadvantage (this phrase was mentioned in the textbook and when this was last asked it was on an open-book exam, so lots of people copied this phrase as their answer). Some answers mentioned that it's easier to access individual characters in a `char[]` than a String, but this isn't really the case (you can use `.charAt()` in a `String`).

For part (ii), many answers said that it was more efficient to look up elements in a `HashMap` than an `ArrayList`, but this isn't the case (`ArrayList` supports lookup by index efficiently through `.get()`). Additionally, some answers said that you could associate multiple values with a single key in a `HashMap` but not an `ArrayList`, which isn't the case.

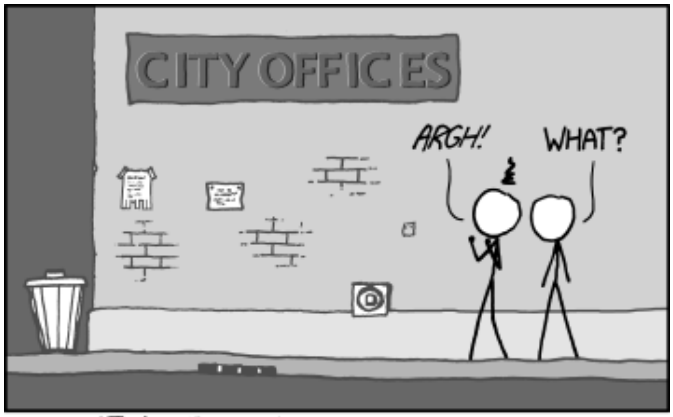
For part (iii), some answers identified that there were no braces in the method, but didn't identify why specifically this would cause the method to always return true. The specific issue was the duplicated line not being part of the nearest if statement, not a lack of braces in general.

Problem Four: Kerning (10 Points)

When I first gave this problem out on the Winter 2012 final exam, it was the most algorithmically challenging question on the exam. Perhaps the easiest way to solve this problem is to verbatim copy one image into the resulting image, then to copy just the black pixels from the second image into the result. A more difficult approach is to break the image into left, middle, and right pieces, copy the left and right verbatim, then blend the middle appropriately. The math here is a bit tricky either way. Below is code for the two solutions:

```
private boolean[][] kern(boolean[][] first, boolean[][] second, int kern) {
      /* Determine the size of the new image. */
      int numRows = first.length;
      int numCols = first[0].length + second[0].length - kern;

      /* Allocate the result. */
      boolean[][] result = new boolean[numRows][numCols];

      /* Copy the first image in verbatim. */
      for (int i = 0; i < first.length; i++) {
            for (int j = 0; j < first[i].length; j++) {
                  result[i][j] = first[i][j];
            }
      }

      /* Copy just the black pixels of the second image offset by some amount. */
      for (int i = 0; i < second.length; i++) {
            for (int j = 0; j < second[i].length; j++) {
                  /* If the pixel is black, copy it. */
                  if (second[i][j])
                        result[i][first[0].length - kern + j] = true;
            }
      }

      return result;
}
```

```
private boolean[][] kern(boolean[][] first, boolean[][] second, int kern) {
      /* Determine the size of the new image. */
      int numRows = first.length;
      int numCols = first[0].length + second[0].length - kern;

      /* Allocate the result. */
      boolean[][] result = new boolean[numRows][numCols];

      /* Copy the first piece of the first image in verbatim. */
      for (int i = 0; i < first.length; i++) {
            for (int j = 0; j < first[i].length - kern; j++) {
                  result[i][j] = first[i][j];
            }
      }

      /* Copy the second piece of the second image in verbatim */
      for (int i = 0; i < second.length; i++) {
            for (int j = kern; j < second[i].length; j++) {
                  result[i][first[0].length - kern + j] = second[i][j];
            }
      }

      /* Merge the two pieces together. */
      for (int i = 0; i < first.length; i++) {
            for (int j = 0; j < kern; j++) {
                  if (first[i][first[0].length - kern + j] || second[i][j])
                        result[i][first[0].length - kern + j] = true;
            }
      }

      return result;
}
```



IF YOU REALLY HATE SOMEONE, TEACH THEM TO RECOGNIZE BAD KERNING.

(xkcd)

Computerized typesetting has a fascinating history. Some major pioneers in the computer science, most notably Don Knuth, spent a lot of time and effort developing software to draw text clearly and elegantly. Steve Jobs is famous for investing effort to make the fonts on Apple computers look as elegant as possible. In fact, in his commencement address here at Stanford in 2005, Jobs specifically mentioned typography and making elegant fonts:

> Reed College at that time offered perhaps the best calligraphy instruction in the country. Throughout the campus every poster, every label on every drawer, was beautifully hand calligraphed. Because I had dropped out and didn't have to take the normal classes, I decided to take a calligraphy class to learn how to do this. I learned about serif and san serif typefaces, about varying the amount of space between different letter combinations, about what makes great typography great. It was beautiful, historical, artistically subtle in a way that science can't capture, and I found it fascinating.
>
> None of this had even a hope of any practical application in my life. But ten years later, when we were designing the first Macintosh computer, it all came back to me. And we designed it all into the Mac. It was the first computer with beautiful typography. If I had never dropped in on that single course in college, the Mac would have never had multiple typefaces or proportionally spaced fonts. And since Windows just copied the Mac, it's likely that no personal computer would have them. If I had never dropped out, I would have never dropped in on this calligraphy class, and personal computers might not have the wonderful typography that they do. Of course it was impossible to connect the dots looking forward when I was in college. But it was very, very clear looking backwards ten years later.
>
> Again, you can't connect the dots looking forward; you can only connect them looking backwards. So you have to trust that the dots will somehow connect in your future. You have to trust in something — your gut, destiny, life, karma, whatever. This approach has never let me down, and it has made all the difference in my life.

(Source: http://news.stanford.edu/news/2005/june15/jobs-061505.html)

On a less serious note: the word “keming” (with an 'm') is sometimes used to mean “excessive kerning.” Also, go and do a Google search for the word “kerning.” It's hilarious.

Problem Five: Animal Hipsters (10 Points)

```
private ArrayList<String>
findAnimalHipsters(HashMap<String, ArrayList<String>> network,
                   HashMap<String, String> favoriteAnimals) {

    ArrayList<String> result = new ArrayList<String>();
    for (String person: network.keySet()) {
        boolean isHipster = true;
        for (String friend: network.get(person)) {
            if (favoriteAnimals.get(person).equals(favoriteAnimals.get(friend)) {
                isHipster = false;
            }
        }
        if (isHipster) result.add(person);
    }
    return result;
}
```

Why we asked this question: We asked this question for a few reasons. First, we thought that this would be a good test of the mechanics of `HashMap`s and `ArrayList`s and would test whether you understood how social networks can be represented in code. Second, we wanted you to have to play around with an example involving several different `HashMap`s whose contents needed to be correlated together. Finally, we wanted to expose you to this particular problem because it's related to a famous problem in computer science called *graph coloring*, which has many interesting theoretical properties.

Common mistakes included incorrectly looping over all the people in the graph, comparing strings using `==` instead of `.equals()`, mixing up the keys and values in the maps, and checking whether someone was a hipster by seeing if *anyone* in the network had the same favorite animal (and not just the person's friends).