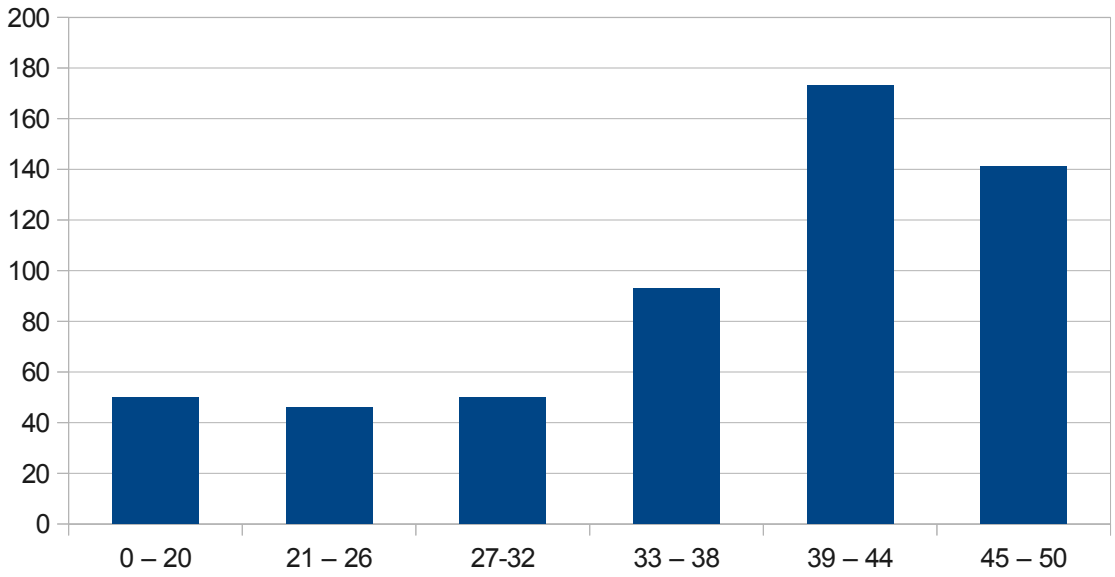
Second CS106A Midterm Exam Solutions

The grade distribution for this exam was as follows:



Overall, the final statistics were as follows:

75th Percentile: 44 / 50 (88%)
50th Percentile: 39 / 50 (78%)
25th Percentile: 28 / 50 (56%)

We are **not** grading this course using raw point totals and will instead be grading on a (fairly generous) curve. Roughly speaking, the median score corresponds to roughly a B+. As always, if you have any comments or questions about the midterm or your grade on the exam, please don't hesitate to drop by office hours with questions! You can also email Vikas or Keith with any questions.

If you think that we made any mistakes in our grading, please feel free to submit a regrade request to us. Just write a short (one-paragraph or so) description of what you think we graded incorrectly, staple it to the front of your exam, and hand your exam to either Vikas or Keith by Wednesday, March 19 at 3:15PM.

Problem One: 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 Two: Accidental Haiku (10 Points)

There are many solutions to this problem. Most of them start off with something like this:

```
private ArrayList<String> findAccidentalHaiku(ArrayList<String> sentences) {
    ArrayList<String> result = new ArrayList<String>();
    for (String sentence: sentences) {
       if (isHaiku(sentence)) result.add(sentence);
    }
    return result;
}
```

The majority of the variability in this problem comes from how people chose to determine whether a sentence was a haiku. Here's a few possible solutions:

```
private boolean isHaiku(String sentence) {
    int syllables = 0;
    boolean l1 = false, l2 = false, l3 = false;
    for (String token: tokenize(sentence)) {
       if (Character.isLetter(token.charAt(0))) {
           syllables += syllablesInWord(token);
           if (syllables == 5) l1 = true;
           if (syllables == 5 + 7) l2 = true;
           if (syllables == 5 + 7 + 5) l3 = true;
       }
    }
    return l1 && l2 && l3 && syllables == 5 + 7 + 5;
}
```

```
private boolean isHaiku(String sentence) {
    int syllables = 0;
    for (String token: tokenize(sentence)) {
       if (Character.isLetter(token.charAt(0))) {
          if (syllables < 5) {
              syllables += syllablesInWord(token);
              if (syllables > 5) return false;
          } else if (syllables < 5 + 7) {
              syllables += syllablesInWord(token);
              if (syllables > 5 + 7) return false;
          } else if (syllables < 5 + 7 + 5) {
              syllables += syllablesInWord(token);
              if (syllables > 5 + 7 + 5) return false;
          } else return false;
       }
    }
    return true;
}
```

```
private boolean isHaiku(String sentence) {
    int s1 = 0, s2 = 0, s3 = 0;
    for (String token: tokenize(sentence)) {
       if (Character.isLetter(token.charAt(0))) {
           if (s1 < 5) {
               s1 += syllablesInWord(token);
           } else if (s2 < 7) {
               s2 += syllablesInWord(token);
           } else {
               s3 += syllablesInWord(token);
           }
       }
    }
    return s1 == 5 && s2 == 7 && s3 == 5;
}
```

Why we asked this question: The initial setup for the question, returning a list of all the sentences that were haiku, was designed to test whether you could iterate over an `ArrayList` correctly. The main logic in this problem boils down to the logic to test whether a sentence is a haiku. There are *many* different ways to do this, but nothing we'd covered in the class so far exactly matched this. We wanted to see if you could use the techniques we'd covered so far in a novel way.

Additionally, we included this problem so that you could get a better appreciation for modular design and code reuse. When you wrote the `tokenize` and `syllablesInWord` method in Assignment 4, they were designed to be used in an entirely different context than the one given here. However, they can be used to solve all sorts of other problems as well.

Common mistakes: The most common mistakes we saw on this problem were forgetting to check whether a sentence could be split into lines of 5-7-5 syllables, or forgetting to account for the case where just the beginning of the sentence could be split into a 5-7-5 format (for example, if there are more than 17 syllables).

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 lots of 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 (this was in the textbook) but did not elaborate on why this was an advantage or disadvantage. 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 always 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: Detecting Speeders (10 Points)

```
private boolean definitelySped(double[] boothPositions,
                               double[] times,
                               double speedLimit) {
    for (int i = 1; i < boothPositions.length; i++) {
        double distance = boothPositions[i] - boothPositions[i - 1];
        double time     = times[i] - times[i - 1];
        if (distance / time > speedLimit) return true;
    }
    return false;
}

private boolean[] findSpeeders(double[] boothPositions,
                               double[][] times,
                               double speedLimit) {
    boolean[] result = new boolean[times.length];
    for (int row = 0; row < times.length; row++) {
        result[row] = definitelySped(boothPositions, times[row], speedLimit);
    }
    return result;
}
```

Why we asked this question: I included this question for two reasons. First, at a high level, I wanted you to get a sense for just how easy it is to do some important analyses on data sets. It doesn't take a lot of code to detect anomalies, as long as you know where to look for them.

Part (i) of this problem was designed to see if you could correctly iterate over the arrays in parallel with one another without walking off the beginning or end of the array. It was also designed to test how you'd handle arrays of size zero or size one; if the method was written correctly, no corrective action is required. Part (ii) of this problem was designed to test your understanding of multidimensional arrays. We hoped that if you had internalized that multidimensional arrays really are “arrays of arrays,” then you'd realize that there isn't much work to be done in the method.

Common mistakes: Many solutions had indexing errors that would cause them to read off the ends of one of the two arrays. Some other solutions would return false as soon as any interval was found where the case wasn't speeding, rather than waiting until the end to check for this. For part (ii), some solutions forgot to create an array of booleans to hold the result, instead just using a single boolean. There was also some confusion about how to get a single row out of the times array in part (ii), with some solutions using the wrong syntax and others inadvertently passing the entire array into `definitelySped`.

Problem Five: The Friendship Paradox (10 Points)

```
private double friendshipGap(HashMap<String, ArrayList<String>> network,
                             String person) {
    int friendsOfFriends = 0;
    int friends = network.get(person).size();
    for (String friend: network.get(person)) {
        friendsOfFriends += network.get(friend).size();
    }
    return (double)friendsOfFriends / friends - friends;
}
```

Why we asked this question: The solution code here is short but dense. You need to extract `ArrayLists` from `HashMaps` and call the right methods on them to determine how many friends each friend has, on average. We also included this question to test whether you remembered that integer division rounds down.

I also included this question as an interesting example of the sorts of results that often arise in social networking analysis. Intuitively, this result is true because people with a huge number of friends will be counted more than people with a small number of friends, so they will upwardly skew the average number of friends each of your friends will have.

Common mistakes: Many solutions iterated over the entire network to get the total average number of friends, rather than just the number of friends of the specified person. When iterating over the specified person's friends, many solutions tried to look people up in the network by using the *index* of that friend rather than the name. There were also many solutions with minor errors involving integer division, such as casting the result of the division rather than one of the operands.