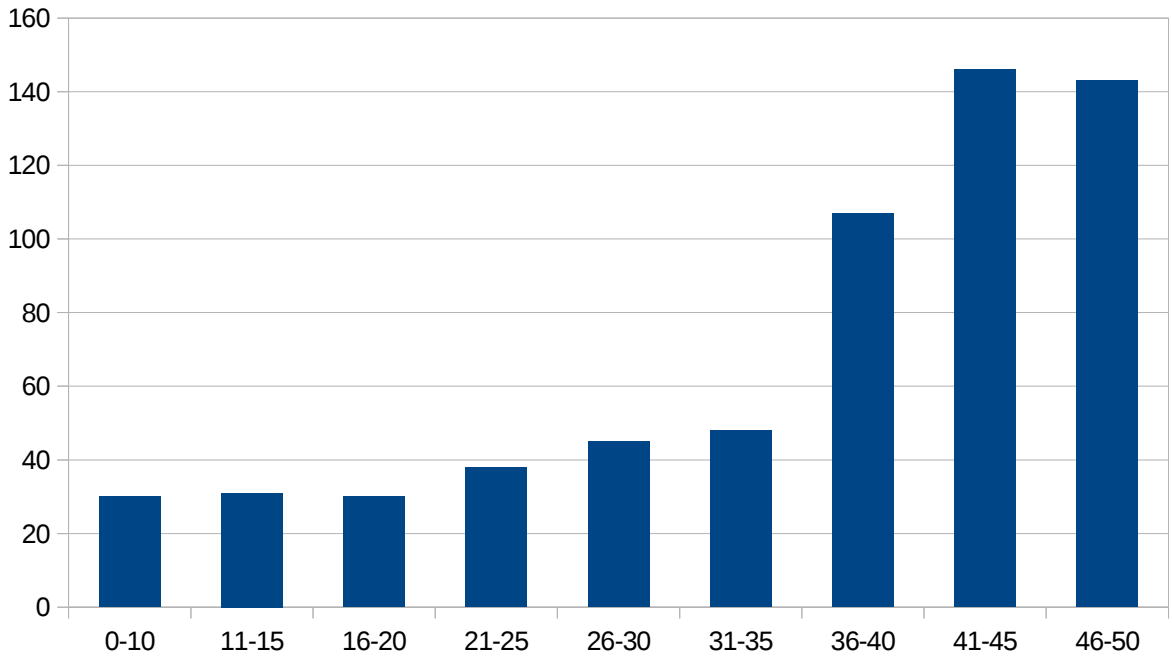
Second CS106A Midterm Exam Solutions

Below is the overall grade distribution for the second CS106A midterm exam:



Because this distribution is slightly skewed, we've historically found that the mean and standard deviation are not particularly good measures of your overall performance on the exam. Instead, we recommend looking at the exam quartiles and getting a rough sense of your performance:

75th Percentile: 45 / 50 (90%)
50th Percentile: 40 / 50 (80%)
25th Percentile: 29 / 50 (58%)

We are not grading this course using raw point totals. Instead, we grade on a (fairly generous) curve where the median score ends up roughly corresponding to a B+.

If you think that we made any mistakes in our grading, please feel free to submit a regrade request to us. To submit a regrade, please fill out a regrade request form (available online from the CS106A website), fill it out, and hand a copy of that form, along with your exam, to Keith or Alisha either during office hours or at lecture. All regrade requests must be received no later than **Tuesday, March 17 at 8:30AM**. (SCPD students will have slightly longer than this, and we'll send out instructions about SCPD regrades over email.)

The solutions in this handout are just one possible set of solutions and represent the cleanest answers that we could think of. Don't worry if your answers don't match what we have here or are more complex than our answers – we had a week to clean up and polish these answers and could check our work on a computer.

Problem One: Initial Casing (10 Points)

```
private String toInitialCase(String input) {
   String result = "";
   for (int i = 0; i < input.length(); i++) {
      if (i != 0 &&
         (input.charAt(i-1) == '\'' || Character.isLetter(input.charAt(i-1)))) {
         result += Character.toLowerCase(input.charAt(i));
      } else {
         result += Character.toUpperCase(input.charAt(i));
      }
   }
   return result;
}
```

Why We Asked This Question: This question was designed to see if you were comfortable doing character-by-character string manipulation. I specifically chose this question because it has similar structure to the `isSyllable` method from Assignment 5 – you need to process characters differently based on what precedes them – and I hoped that most of you would feel comfortable writing similar code on the exam. Plus, I wanted to find some way to sneak some words of encouragement into the exam itself, and the sample inputs to this method seemed like a good place to put them. ☺

Common Mistakes: The main challenge we saw people struggle with in this problem was handling the very first character of the string. Some solutions incorrectly would walk off the front of the string, and others would start their loops at position one but mishandle the first character.

Many solutions had the wrong syntax for using the character manipulation methods. We saw lots of solutions that tried to call methods on variables of type `char`, for example.

We saw many pieces of code that forgot that strings were immutable and that it was necessary to build up a new string to hold the result. Sometimes people forgot entirely to append the characters to a new string, and sometimes people tried adding them back into the original string, usually causing some sort of infinite loop. We also saw many solutions that assumed that `Character.toUpperCase` or `Character.toLowerCase` would transform the input character they received as an argument, even though that's not possible in Java (don't forget about pass-by-value!)

Problem Two: Jumbled Java hiJinks (10 Points Total)

There are many possible correct answers to these problem. If your answer doesn't match ours, don't worry! You very well may still be correct.

Implementation A **(2 Points)**

```
private boolean areAllValuesEqual(int[] arr) {
1:   for (int i = 0; i < arr.length; i++) {
2:      if (arr[i] != arr[i + 1]) {
3:         return false;
4:      }
5:   }
6:   return true;
}
```

Give a non-`null` input array that causes implementation A to behave incorrectly.

[137]

Why does implementation A fail when given this input?

The final iteration of the loop will try to read past the end of the array, triggering an `ArrayIndexOutOfBoundsException`. Any nonempty input array will cause this behavior.

Implementation B **(3 Points)**

```
private boolean areAllValuesEqual(int[] arr) {
 1:   boolean allSame = true;
 2:
 3:   for (int i = 0; i < arr.length; i++) {
 4:      if (arr[i] == arr[0]) {
 5:         allSame = true;
 6:      } else {
 7:         allSame = false;
 8:      }
 9:   }
10:
11:   return allSame;
}
```

Give a non-`null` input array that causes implementation B to behave incorrectly.

[137, 42, 137]

Why does implementation B fail when given this input?

On each iteration of the outer `for` loop, the inner `if` statement will update `allSame` to whether the currently-read value is equal to the first value. This means that after the loop finishes, the value of `allSame` will be `true` if the first and last elements of the array match and `false` otherwise. Given an array where all values are not the same but where the first and last values match, the method will return `true` instead of returning `false`.

Implementation C **(2 Points)**

```
private boolean areAllValuesEqual(int[] arr) {
1:   int total = 0;
2:
3:   for (int i = 0; i < arr.length; i++) {
4:      total += arr[i];
5:   }
6:
7:   return total / arr.length == arr[0];
}
```

Give a non-**`null`** input array that causes implementation C to behave incorrectly.

There are many options: [], [0, 1, -1], and [2, 2, 3] all exercise different errors in the code.

Why does implementation C fail when given this input?

If the input array is empty, this will divide by zero and trigger an exception rather than returning `true`. If the input array has several different values whose average is the first number, then the method will return `true` instead of returning `false`. Finally, if the input array consists of different values whose average is *not* equal to the first value, but is if you use integer division, the method will return `true` instead of returning `false`.

Implementation D **(3 Points)**

```
private boolean areAllValuesEqual(int[] arr) {
 1:    int min = 0;
 2:    int max = 0;
 3:
 4:    for (int i = 0; i < arr.length; i++) {
 5:       if (arr[i] > max) {
 6:          max = arr[i];
 7:       }
 8:       if (arr[i] < min) {
 9:          min = arr[i];
10:       }
11:    }
12:
13:    return min == max;
}
```

Give a non-**`null`** input array that causes implementation D to behave incorrectly.

[1]

Why does implementation D fail when given this input?

This code does not properly find the minimum and maximum values in the array. Specifically, the logic in the two internal if statements will only update `min` to the smallest *nonpositive* number and `max` to the largest *nonnegative* number because `min` and `max` default to 0. Therefore, any input array whose values are all positive or all negative will cause the method to return `false` even if the values are the same because `min` and `max` will have the wrong values.

Why We Asked This Question: This question was designed to see whether you were comfortable reading someone else's code and, in doing so, to find the bugs in it. Reading and writing code that you yourself didn't write is a very useful skill and makes it significantly easier to debug your own code, and we wanted to assess how well you were able to do this.

Common Mistakes: Most of the answers that were submitted here were correct. For part (i), many answers incorrectly claimed that, since the last line of the method is `return true`, the method itself must always return `true`. We also saw many answers that noticed something was fishy about the last array access, but incorrectly claimed that the effect of reading off the end of an array is to read `null` rather than to cause a crash. For parts (ii) and (iii), some answers argued that in an array with negative indices, the methods wouldn't notice the elements before index 0 and therefore wouldn't work correctly. This isn't a problem since, in Java, arrays can't have negative indices.

Problem Three: *k*-Grams (10 Points)

```
private ArrayList<String[]> kGramsIn(ArrayList<String> text, int k) {
    ArrayList<String[]> result = new ArrayList<String[]>();
    for (int i = 0; i < text.size() - k + 1; i++) {
       String[] kGram = new String[k];
       for (int j = 0; j < k; j++) {
          kGram[j] = text.get(i + j);
       }
       result.add(kGram);
    }
    return result;
}
```

Why We Asked This Question: This question was designed to assess several specific skills. First, we wanted to see if you were comfortable working with `ArrayList`s of arrays, something that we didn't really talk about in this course but which behaves as you'd probably expect it to. Second, we wanted to see if you would be able to iterate over the proper subrange of the master `ArrayList` in each step; the loop bounds here are tricky! Finally, we wanted to see how you'd fill in each *k*-gram, which involved iterating over just a subrange of the original string.

On top of all this, we wanted you to get some exposure to *k*-grams, which show up all the time in machine learning and computational linguistics. They're frequently used in places where we want to learn more about the context in which a word is used – by getting *k*-grams from text for reasonably-sized *k* (say, 5 or 7), we can preserve some amount of the context in which each word appears without having to fully understand what the entire text says. You also see *k*-grams used in machine translation for the same reason – if you pair up *k*-grams from a text in one language with *k*-grams from the same text in a different language, you can (ideally) use the local context around each word in each language to try to learn how to translate words and grammatical structures. Both of these approaches are actually used in industrial software systems.

Common Mistakes: A large number of solutions tried to represent each *k*-gram as a single `String` rather than an array of strings. Although this is conceptually not that far from the right answer, it produces output in a different format than expected and ends up avoiding some of the tricky array gymnastics we were hoping to exercise in this problem. Many solutions had the wrong indices in the top-level for loop, either by forgetting to stop early at all or trying to stop early but doing so incorrectly.

The most common mistake on this problem probably was allocating the arrays to hold each *k*-gram in the wrong place. Many solutions created the array outside of both loops or inside the innermost loop. The former will work incorrectly because it will destructively modify previously-created *k*-grams, and the latter will work incorrectly because it either leads to a scoping error or results in the *k*-gram getting added too early.

Finally, many solutions had the wrong logic for reading from the original `ArrayList` or in writing back to the *k*-gram array. The indices being read or written to are not the same in each case, which tripped a lot of people up. Many solutions tried to fix this by adding in a new variable representing the write location, but scoped it incorrectly and ended up reading or writing values in the wrong place.

Problem Four: Andy Warhol Paintings (10 Points)

```
private GImage andyWarholize(GImage source, int[] palette) {
    int[][] pixels = source.getPixelArray();
    for (int row = 0; row < pixels.length; row++) {
       for (int col = 0; col < pixels[row].length; col++) {
           pixels[row][col] = closestMatchFor(pixels[row][col], palette);
       }
    }
    return new GImage(pixels);
}

private int closestMatchFor(int color, int[] palette) {
   int bestMatch = palette[0];
   for (int i = 1; i < palette.length; i++) {
      if (similarityOf(color, palette[i]) > similarityOf(color, bestMatch)) {
          bestMatch = palette[i];
      }
   }
   return bestMatch;
}
```

Why We Asked This Question: We chose to include this question for several reasons. First, the top-level structure of the problem requires you to work with multiple different arrays of different dimensions simultaneously, a skill that's important in general software design but can be surprisingly tricky (this is one of the reasons why you may have found the Tone Matrix assignment difficult). Second, I wanted to include at least one image processing question on the exam so that you could get a better sense for how to programmatically create and modify images. Finally, this method requires you to find the best matching object out of a list of objects, which is a fairly common programming idiom (and hopefully something you saw in the Tone Matrix).

Independently, I just thought this was a really fun program to write. I spent a lot of time taking some of my old photos and Warholizing them when preparing this exam and ended up sending a few of the better results to my friends and family. If you get a chance, try coding this one up! It's really fun! I ended up generating a palette of four random colors each time I called the `warholize` method, and while some palettes don't work very well, you'll pretty frequently get some really great results.

Common Mistakes: Many solutions to this problem tried to decompose each color into its individual red, green, and blue components, which wasn't actually necessary in this problem and (usually) led to incorrect solutions. Similarly, many solutions found the best matching color, but then tried to construct a new pixel color from it unnecessarily.

The other main spot where we saw problems was in the logic to find the best matching color. Some solutions ended up conflating the best matching color and its similarity or storing the "closeness" of the best color as an `int` rather than a `double`. A fair number of solutions also tried to find the best matching color only by comparing adjacent elements in the palette array, rather than by looking for the largest overall color.

Finally, as in Problem Three, we saw lots of issues due to improperly-scoped variables. Many solutions correctly introduced variables to store the best matching color or its similarity to the current pixel, but scoped them too deeply in a loop nest to be used later on or too high up, causing values from one iteration to carry over into the next.

Problem Five: Finding Coworkers (10 Points)

```
private boolean areAtSameCompany(String p1, String p2,
                                 HashMap<String, String> bosses) {
   return ceoFor(p1, bosses).equals(ceoFor(p2, bosses));
}

private String ceoFor(String person, HashMap<String, String> bosses) {
   while (bosses.containsKey(person)) {
      person = bosses.get(person);
   }
   return person;
}
```

Why We Asked This Question: I chose to include this question on the exam for several reasons. First, I wanted to include a question that tested your understanding of how to look up keys and values inside a `HashMap`. The core of the solution involves repeatedly doing both. Second, I wanted to give you practice working with graph structures in Java. Although this `HashMap` doesn't look like the ones we saw in class when studying graphs, the `HashMap` does define a graph (specifically, a *reverse-directed rooted tree*) and the operations you need to perform in the course of writing this method are essentially walks over the graph. Finally, I wanted to test whether you remembered how to check whether two strings are equal, something we haven't seen in a while.

Although this question was phrased in terms of people working at a company and finding CEOs, this problem was originally based on a specialized data structure called a *disjoint-set forest* that's used as a subroutine in several interesting graph algorithms. One of them, *Kruskal's algorithm*, is typically covered toward the tail end of CS106B. It's used to find the cheapest way to provide power to a variety of houses, to discover cell differentiation lineages in computational biology, and as a subroutine in algorithms for planning efficient postal delivery routes. If you'd like to learn more, take CS106B!

Common Mistakes: Although the problem description specifically mentions that names are case-sensitive in this problem, we saw a lot of solutions that began by transforming names to lower-case. That's only necessary if you want names to be case-*in*sensitive, which wasn't the case here.

We also saw many solutions that tried to process the `HashMap` by iterating over all its keys rather than by repeatedly calling the `.get()` method. This wasn't necessary here, and generally speaking isn't something that you commonly see used in a `HashMap`.

As with the other problems in this exam, scoping was a common issue in this problem. We saw lots of solutions where important variables were scoped too tightly inside a loop or too broadly outside a loop, causing compile-time or runtime errors.

Many solutions included some special cases for when one or both of the input people were the CEOs of their companies. This isn't strictly necessary and in many cases actually introduced bugs that wouldn't have been present were the special-case code removed. We also saw a lot of people forget to compare strings with the `.equals` method, instead opting for `==`.

Finally, we saw many solutions that sat in a loop trying to replace each person with their CEO, but which ended up ending the loop as soon as one person's CEO was found instead of both peoples' CEOs. This caused problems in cases where the inputs weren't at the same levels of their respective corporate hierarchies.