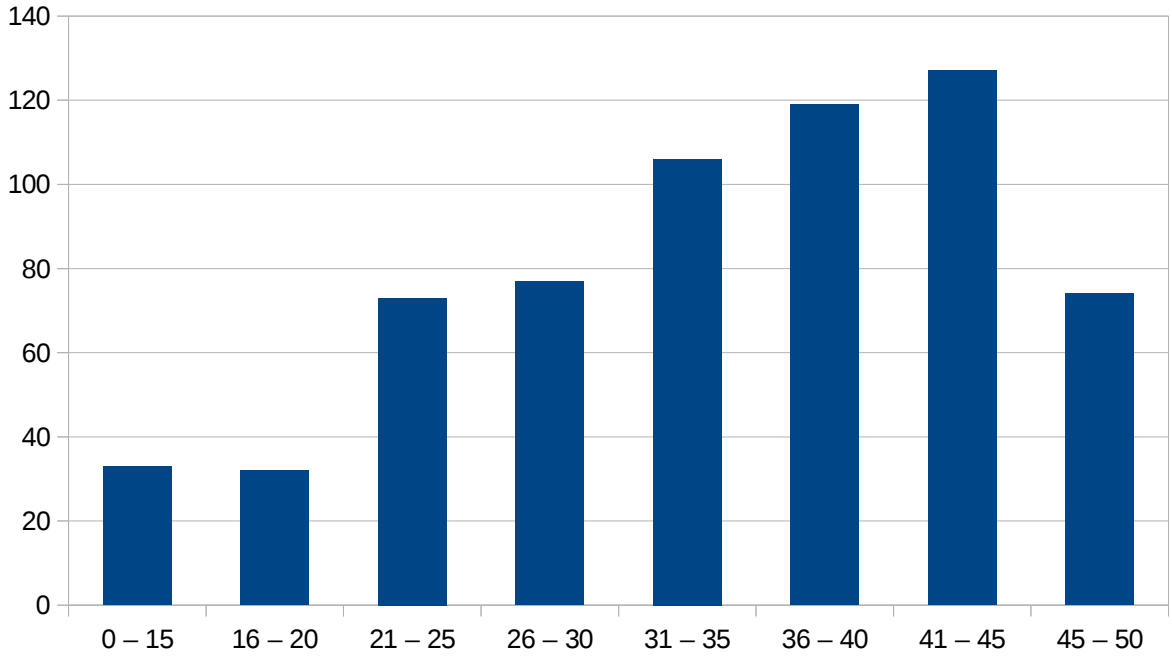
CS106A Midterm Exam Solutions

Below is the overall grade distribution for the first 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: 42 / 50 (84%)
50th Percentile: 35 / 50 (70%)
25th Percentile: 27 / 50 (54%)

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 **Wednesday, February 25** at 4:15PM. (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: `TrainConductorKarel` (10 Points)

```
import stanford.karel.*;

public class TrainConductorKarel extends SuperKarel {
    public void run() {
       for (int i = 0; i < 4; i++) {
          processOneSide();
          turnLeft();
       }
       handleLastBeeper();
    }

    private void processOneSide() {
       while (frontIsClear()) {
          if (beepersInBag()) {
              putBeeper();
          } else if (beepersPresent()) {
              pickBeeper();
          }
          move();
       }
    }

    private void handleLastBeeper() {
       if (beepersInBag()) {
           putBeeper();
       }
    }
}
```

Why we asked this question: We asked this question primarily to test your ability to decompose a problem down into smaller pieces and, in doing so, to keep track of a consistent set of preconditions and postconditions for each piece. It's quite difficult to solve this problem if you don't set up clear conditions for what each method should do and the relative orientation and position of Karel at each step.

Common mistakes: Most solutions that we saw to this problem were on the right track (no pun intended). There were several common classes of mistakes that we ended up seeing on this problem.

Many solutions ended up putting Karel into an infinite loop, constantly moving beepers around the world, or otherwise ended up processing too many sides of the world. Sometimes this resulted from putting the entire program into the wrong top-level loop structure (`while` instead of `for`, for example), and sometimes it resulted from code that sometimes turned Karel around a corner when moving around the world and sometimes didn't.

Many solutions inadvertently forgot to move Karel off of the beeper Karel just finished moving forward a step. This would cause Karel to pick up a beeper, drop it, then pick it back up again and carry it around the world. In many cases, it's difficult to fix this bug without introducing some other errors into the program, especially due to the complexity of navigating beepers at the end of a row.

Some solutions had trouble getting Karel to move down a side of the world and find the next beeper. We saw many solutions where Karel would crash into a wall searching for a beeper or only process the very first beeper in a row, often accidentally.

Finally, we saw many solutions that worked correctly in most cases, but would incorrectly handle the very first, very last, or (often) the second-to-last beeper on each side of the world.

Problem Two: Mirror Box Drawings (10 Points)

```
import acm.program.*;
import acm.graphics.*;
import java.awt.event.*;

public class MirrorBox extends GraphicsProgram {
    /* The radius of each circle that should be drawn. */
    private static final double RADIUS = 5.0;

    public void run() {
       drawCenterLine();
       addMouseListeners();
    }

    private void drawCenterLine() {
       double centerX = getWidth() / 2.0;
       add(new GLine(centerX, 0, centerX, getHeight()));
    }

    public void mousePressed(MouseEvent e) {
       handleMouse(e.getX(), e.getY());
    }

    public void mouseDragged(MouseEvent e) {
       handleMouse(e.getX(), e.getY());
    }

    private void handleMouse(double x, double y) {
       drawCenteredCircle(x, y);
       drawCenteredCircle(getWidth() - x, y);
    }

    private void drawCenteredCircle(double x, double y) {
       GOval circle = new GOval(x - RADIUS, y - RADIUS, 2 * RADIUS, 2 * RADIUS);
       circle.setFilled(true);
       add(circle);
    }
}
```

Why we asked this question: We asked this question for several different reasons. First, we wanted to see if you were comfortable writing programs that required multiple different mouse handlers, something that you've seen in section problems and in lecture but not in the assignments. Second, we hoped the question would give you a chance to demonstrate your ability to work through the mathematics of centering objects and reflecting objects across the window. Finally, we honestly thought that this program was a lot of fun to play with, and wanted to share it with you during the exam!

Common mistakes: The most common mistakes we encountered in this question involved centering and positioning the circles on the screen. Many solutions forgot to center the two circles or incorrectly computed the coordinates of the second circle from the first. The math involved in reflecting a circle across the center line isn't too complex once you see it, but it's not immediately obvious.

We also saw several mistakes with the event handlers. Some solutions put the code for `mouseDragged` inside of a while loop, which usually resulted in the program not working correctly. There's no need for a loop here because `mouseDragged` is automatically called every time the mouse is dragged somewhere in the window. Other solutions only had one of the two necessary mouse handlers, or incorrectly tried to unify the two handlers together into one. We frequently saw handlers that were marked `private` rather than `public`, and in some cases saw scoping errors where code in one handler tried to access data available in the other.

Problem Three: RNA Hairpins (10 Points)

```
private boolean isRNAHairpin(String rna) {
   if (rna.length() < 4 || rna.length() % 2 != 0) {
      return false;
   }
   for (int i = 0; i < rna.length() / 2 - 1; i++) {
      if (text.charAt(i) != matchOf(text.charAt(text.length() - 1 - i))) {
         return false;
      }
   }
   return true;
}

private char matchOf(char ch) {
   if (ch == 'A') return 'U';
   if (ch == 'C') return 'G';
   if (ch == 'G') return 'C';
   return 'A';
}
```

Why we asked this question: We chose to include this question for a few reasons. First, this question tested your ability to iterate over different ranges of strings and your ability to set the bounds of different loops correctly. Second, this question assessed whether you were comfortable working with strings one character at a time. Finally, we thought that many of you might not have heard of RNA hairpins before and thought it would be fun to share a new tidbit or two during the exam. ☺

Common mistakes: Many solutions tried to detect whether the input string was too short or had odd length, but ended up doing so incorrectly (we saw many solutions that returned `false` if the length was too short *and* the length was odd, rather than returning `false` if the length was too short *or* the length was odd.)

A decent number of solutions didn't understand what the parameter to this method meant and ended up using `readLine` to read in the line of text to process. Remember that methods that take in parameters aren't responsible for actually getting the values to those parameters – the caller will specify these values for you. Similarly, many solutions used `println` to print out the result rather than using `return` to return it.

We saw a fair number of off-by-one errors in the loop bounds. Some solutions tried to read off the end of the string by getting the index of the last character wrong, or ended up having a loop that accidentally included the middle two characters.

Many solutions inadvertently returned from the method too early by putting code into the body of a `for` loop that returned `true` if the characters matched and `false` otherwise. This cuts off the search too early and will say "yes" to any strand whose first and last characters match.

Finally, we saw many solutions in which the logic for checking whether characters matched only at a few of the possible matches.

Problem Four: Jumbled Java hiJinks (10 Points Total)

(i) Solve for *x* **(5 Points)**

```
/* Program A */
int x = readInt();
if (0 < x || x < 0) {
    println("success");
}
```

Any number ***except for*** 0 works.

Answer for Program A: 137

```
/* Program B */
int x = readInt();
if (x * 4 / 5 == x * (4 / 5)) {
    println("failure");
}
println("success");
```

The right-hand quantity is always 0 due to integer division. The left-hand quantity is nonzero as long as |x| > 2. Any number ***except for*** -1, 0, or 1 works.

Answer for Program B: 137

```
/* Program C */
int x = readInt();
if (x < 0 || x >= 10) {
   println("failure");
}
while (x > 0 && x < 10) {
    x *= 2;
}
if (x >= 19) {
    println("success");
}
```

Any input number less than 0 or greater than 9 automatically fails. That leaves numbers in the range from 0 to 9, inclusive. Since the `while` loop stops as soon as `x` becomes greater than 10, the maximum possible value for `x` on entry to the loop is 9, so the maximum possible value for `x` on exit from the loop is 18. Therefore, there is no solution.

Answer for Program C: *no solution*

```
/* Program D */
int x = readInt();
if (x >= 30) {
    println("failure");
}
if (x >= 20) {
    println("success");
}
if (x >= 10) {
    println("failure");
}
```

The only way to print `success` is to pick a number greater than or equal to 20, but then `failure` is also printed by the next `if` statement.

Answer for Program D: *no solution*

```
/* Program E */
int x = readInt();
for (int j = 137; j > x; j--) {
    println("failure");
}
println("success");
```

Any number greater than or equal to 137 works.

Answer for Program E: 137

(ii) Program Tracing **(5 Points)**

```
import acm.program.*;

public class PaddingtonJava extends ConsoleProgram {
	public void run() {
		int montgomery = 72;
		int lucy = 34;

		println("pastuzo(montgomery) = " + pastuzo(montgomery));
		println("montgomery = " + montgomery);
		println("lucy = " + lucy);

		henry(lucy, montgomery);
		println("montgomery = " + montgomery);
		println("lucy = " + lucy);
	}

	private int pastuzo(int montgomery) {
		montgomery = montgomery % 10 * 10 + montgomery / 10;
		return montgomery;
	}

	private void reginald(String montgomery) {
		montgomery = "marmalade sandwich";
		println("montgomery = " + montgomery);
	}

	private int henry(int montgomery, int lucy) {
		String millicent = montgomery + " pigeons";

		reginald(millicent);
		println("millicent = " + millicent);
		println("montgomery = " + montgomery);

		return lucy + montgomery * 2;
	}
}
```

```
pastuzo(montgomery) = 27
montgomery = 72
lucy = 34
montgomery = marmalade sandwich
millicent = 34 pigeons
montgomery = 34
montgomery = 72
lucy = 34
```

Why we asked this question: Part (i) of this question was designed to test your ability to read code that you yourself hadn't written. It was also designed to exercise various features of Java that can behave counterintuitively, such as sequential `if` statements without intervening `else`s and integer division. Part (ii) of this question was designed to test your understanding of scoping, parameter passing, and operator precedence. It was also designed to see if you understood the mechanics behind string immutability and passing object references by value.

Problem Five: Talkin' 'Bout My Generations (10 Points Total)

Part One: Modeling Child Counts (3 Points)

```
private int randomNumChildren() {
   int result = 0;
   RandomGenerator rgen = RandomGenerator.getInstance();

   while (rgen.nextBoolean()) {
       result++;
   }

   return result;
}
```

Part Two: Simulating Generations (7 Points)

```
private int numGenerations() {
   int thisGeneration = 1;
   int numGenerations = 0;

   while (thisGeneration > 0) {
       numGenerations++;

       int nextGeneration = 0;
       for (int i = 0; i < thisGeneration; i++) {
           nextGeneration += randomNumChildren();
       }

       thisGeneration = nextGeneration;
   }

   return numGenerations;
}
```

Why we asked this question: Part (i) of this question was designed to see if you remembered how to simulate a series of coin flips. It was deliberately structured to mirror the Saint Petersberg Game problem from Assignment 3. We chose to ask part (ii) of this question to see if you could figure out how to model a complicated process that looks very little like what you've seen so far in CS106A using only the tools that you've had access to so far. It requires some insight to see that you just need to track quantities of people in each generation rather than actual ancestor/descendant relationships, and we hoped that the hint would help you figure this out. Finally, we thought it would be fun to share this mathematical model with you. It has a surprisingly large number of applications in computer science!

Common mistakes: In part (i), many solutions had the wrong syntax for getting the random generator. Most answers in this part of the problem were more or less correct.

In part (ii), many solutions correctly tried to figure out how many children would be around in the next generation, but did so in a way that ended up either leaving the generation size at 1 or otherwise overwriting the generation size at the wrong time. It's important to keep track of when the generation size needs to get overwritten and where it should be overwritten.