Problem 1—Short answer (20 points)

Suppose that you have been assigned to take over a project from another programmer who has just been dismissed for writing buggy code. One of the methods you have been asked to rewrite has the following comment and prototype:

```java
private int[] insertValue(int value, int[] array) {
    int[] result = new int[array.length + 1];
    for (int i = 0; i < result.length; i++) {
        result[i] = array[i];
    }
    int pos = 0;
    for (int i = 0; i < array.length; i++) {
        if (value > array[i]) {
            pos = i;
            break;
        }
    }
    for (int i = result.length; i >= pos; i--) {
        result[i] = result[i - 1];
    }
    result[pos] = value;
    return result;
}
```

Unfortunately, the code contains several bugs. Circle the bugs in the implementation and write a short sentence explaining the precise nature of each problem you identify.
Problem 2—Using the acm.graphics library (30 points)

Although the definition of filling for an arc (see page 314 in the text) is not necessarily what you would want for all applications, it turns out to be perfect for the problem of displaying a traditional pie chart. Your job in this problem is to write a method

```java
private GCompound createPieChart(double r, double[] data)
```

that creates a GCompound object that displays a pie chart for the specified data values. The parameter `r` represents the radius of the circle in which the pie chart is displayed, and `data` is the array of data values you want to plot in the chart. The operation of the `createPieChart` method is easiest to illustrate by example. If you execute the `run` method

```java
public void run() {
    double[] data = { 45, 25, 15, 10, 5 };  
    GCompound pieChart = createPieChart(50, data); 
    add(pieChart, getWidth() / 2, getHeight() / 2); 
}
```

your program should generate the following pie chart in the center of the window:

![Pie Chart Image]

The red wedge corresponds to the 45 in the data array and extends 45% around the circle, which is not quite halfway. The yellow wedge then picks up where the red wedge left off and extends for 25% of a complete circle. The blue wedge takes up 15%, the green wedge takes up 10%, and the pink wedge the remaining 5%.

As you write your solution to this problem, you should keep the following points in mind:

- The values in the array are not necessarily percentages. What you need to do in your implementation is to divide each data value by the sum of the elements to determine what fraction of the complete circle each value represents.
- The colors of each wedge are specified in the following constant array:
  ```java
  private static Color[] WEDGE_COLORS = {
      Color.RED, Color.YELLOW, Color.BLUE, Color.GREEN, 
      Color.PINK, Color.CYAN, Color.MAGENTA, Color.ORANGE
  };
  ```
  - If you have more wedges than colors, you should just start the sequence over, so that the eighth wedge would be red, the ninth yellow, and so on.
  - The reference point of the GCompound returned by `createPieChart` must be the center of the circle.
Problem 3—Strings (20 points)
The \textit{rotation} of an English word is the word formed by removing the first letter of the word and then tacking it on to the end of the word. For example, the rotation of the word TABLE is ABLET, since we removed the T from the beginning and appended it to the end. Sometimes rotating a word produces a new word (amazingly, “ablet” is a word for a type of fish), while other times rotating a word doesn't produce a new word (for example, rotating IBEX to BEXI doesn't produce a new word).

In some rare cases, it's possible to rotate a word several times to produce a series of words that end up back at the original word itself. For example, the word EAT can be rotated as

\[
\text{EAT} \rightarrow \text{ATE} \rightarrow \text{TEA} \rightarrow \text{EAT}
\]

Notice that each intermediate word is a real English word. Similarly, the word SPA can be rotated through back to itself:

\[
\text{SPA} \rightarrow \text{PAS} \rightarrow \text{ASP} \rightarrow \text{SPA}
\]

(Yes, “pas” is a word. It's a step in a dance.) However, most words can't be rotated this way, since the intermediary words might not be legal English words. For example, rotating the word “code” gives

\[
\text{CODE} \rightarrow \text{ODEC} \rightarrow \text{DECO} \rightarrow \text{ECOD} \rightarrow \text{CODE}
\]

And “odec” isn't a word (though, interestingly, both “deco” and “ecod” are). If all of the rotations of a word are legal, then we say that a word is a \textit{rotating word}.

Write a method

\[
\text{private Set<String> findRotatingWords(Set<String> dictionary)}
\]

that accepts as input a \texttt{Set<String>} containing all words in English, stored in lower-case, and returns a \texttt{Set<String>} representing all rotating English words. This set would contain “eat” and “spa,” for example, but not “code” or “ibex.”
Problem 4—Arrays (25 points)

If you spent a lot of time playing your Yahtzee program, you know that it would be wonderful if there were a two-pair category for all the times you just missed getting a full house. Write a Java method

```java
private boolean checkTwoPairCategory(int[] dice)
```

that takes an array of five die values and returns `true` if and only if the values in the `dice` array contain two pairs and a fifth value that does not match the others.

In writing your answer to this problem, you should keep the following points in mind:

- You do not have to check that the array contains five elements or that the values in the array fall between 1 and 6.
- You may not change or reorder the values in the array you have been passed by the caller. You may, however, create temporary arrays as part of your implementation.
- A set of dice fits the two-pair category only when the paired values are different and the fifth value doesn’t match any of the others.
Problem 5—Building graphical user interfaces (40 points)

If you'll recall from lecture, each pixel in an image is represented by three values – its red component, green component, and blue component. By altering the values of these components, you can dramatically change the appearance of an image. Setting these values to 0 will produce the effect of looking at the image through a colored filter that blocks out that particular color. For example, if we set the green and blue channels to 0, only the red component of the image will remain, which gives the effect of looking at the image through a red filter.

Your task in this problem is to write a program that allows the user to filter an image by selectively turning off different channels of the image. When the program starts up, the image will appear in true color, with the red, green, and blue channels all enabled, as shown here:

At the bottom of the screen are three checkboxes, one for each component. Clicking on any of these checkboxes toggles whether or not that channel is displayed. For example, clicking the green checkbox will disable the green component, as shown here:

Write a GraphicsProgram that does the following:

- On startup, loads the image specified in the IMAGE_FILENAME constant and adds it to the canvas. (You don't need to center it; assume that the image exactly fits into the window)
- Creates three checkboxes, one for red, green, and blue, all initially checked.
- Upon the user clicking any checkbox, updates the display by redrawing the filtered image. You should filter the image by iterating over the pixels and for each channel (red, green, and blue), setting that channel to 0 if the appropriate checkbox isn't selected.
Problem 6—Using Java collections (25 points)

In the xkcd strip that appeared in the handout for Assignment #2, Randall Monroe makes reference to the hailstone sequence that you learned about back in the Welcome to Java assignment (the Collatz Conjecture to which he refers is the as-yet-unresolved claim that this process always terminates). As the cartoon makes clear, many of the sequences include common patterns. For example, all hailstone sequences (except for 1, 2, 4, and 8, of course) go through the value 16 and therefore share the last four elements of the path. If you’ve followed through this path once, you don’t need to follow it again.

Keeping track of these common patterns can make calculating the length of hailstone sequences vastly more efficient. Suppose, for example, that you are calculating the number of steps in all the hailstone sequences between 1 and 100. What happens when you get to 24, which appears at the lower left corner of the cartoon? By the time you get to this point, you’ve already figured out that it takes nine steps to get from 12 down to 1, so that 24 must take ten steps, given that 12 is just one step away from 24.

It’s even more interesting to follow what happens with 7, which is in the middle of the right hand side. Here, figuring out the number of steps for 7 entails running through the number 22, 11, 34, 17, 52, 26, 13, 40, and 20 before you come to 10, which you’ve already encountered in counting the number of steps from 3 and 6. At this point, you have learned an enormous amount. Given that it takes six steps to reach 1 starting at 10, you know that 20 takes seven steps, 40 takes eight, 13 takes nine, and so forth, backwards along the list of numbers in the chain, until you determine that there are sixteen steps in the hailstone chain between 7 and 1. If you kept track of these values in, for example, a HashMap<Integer,Integer>, you’d already know the answers for the other values in the chain.

Keeping track of previously computed values so that you can use them again without having to recompute them is called caching.

Write a function

```java
private int countSteps(int n, HashMap<Integer,Integer> cache) {

```

that determines the number of steps in the hailstone sequence starting at n. The second parameter is a HashMap containing previously computed values. Your implementation should look up each value it encounters during the process to check whether the answer from that point is already known. If so, it should use that previously computed value not only to compute the current result, but also to add the counts for all the intermediate steps to the HashMap. Thus, if you call countSteps with 7 as the first parameter, your code should add the counts for 20, 40, 13, 26, 52, 17, 34, 11, 22, and 7 to the HashMap before returning the answer.
Problem 7—Essay: Extensions to the assignments (20 points)

In many ways, the assignment that offers the greatest opportunities for extension is the Breakout game from Assignment #3. For this problem, imagine that you are trying to create a “Super Breakout” in which multiple balls can be in play at the same time.

Here is how you would like your new program to work. In the initial layout of bricks at the top of the screen, a random collection of bricks (a different set each time) are designated superbricks and are marked on the display by having a black border instead of being solidly colored throughout. Superbricks are relatively rare; as you lay them out, each brick has a 5% chance of being a superbrick.

The game proceeds exactly as it did before until the ball hits a superbrick, at which point the program fires off another ball just as it did at the beginning of the turn. In each time step of the game, both balls move, bounce off the walls and the paddle, and collide with bricks (and possibly each other). The number of balls increases as you hit more superbricks, and decreases when a ball falls off the bottom. The current turn, however, ends only when there are no balls remaining on the board.

Discuss the changes that would be necessary in the Breakout implementation to implement this extension to the game. Keep in mind that this is an essay question, and you should not feel compelled to write any actual code unless you feel that doing so is the best way to convey your ideas. You should, however, explain as clearly as you can what parts of the program will need to change and what data structures you would need to add to the program to implement this feature.