

Practice Midterm Examination

Midterm Time: Monday, July 24th, 7:00P.M.–9:00P.M.
Midterm Location: Hewlett 200

Based on handouts by Mehran Sahami, Eric Roberts and Patrick Young

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on the midterm examination. We do not guarantee that the number of questions in this sample exam will match the number on the real exam, nor that every kind of problem shown here will exactly match the kinds of problems shown on the final exam (though the real exam will be generally similar overall).

The midterm exam is open-textbook, closed-notes and closed-electronic-device. A “syntax reference sheet” will be provided during the exam (it is omitted here, but available on the course website). It will cover all material presented up to the midterm date itself. Please see the course website for a complete list of midterm exam details and logistics.

General instructions

Answer each of the questions included in the exam. If a problem asks you to write a method, you should write only that method, not a complete program. Write all of your answers directly on the *answer pages provided for that specific problem*, including any work that you wish to be considered for partial credit. Work for a problem not included in a problem’s specified answer pages will not be graded.

Each question is marked with the number of points assigned to that problem. The total number of points is 120. We intend for the number of points to be roughly comparable to the number of minutes you should spend on that problem.

In all questions, if you would like to use methods or definitions implemented in the textbook (e.g. from an example problem), you may do so by giving the name of the method and the chapter number in which that definition appears.

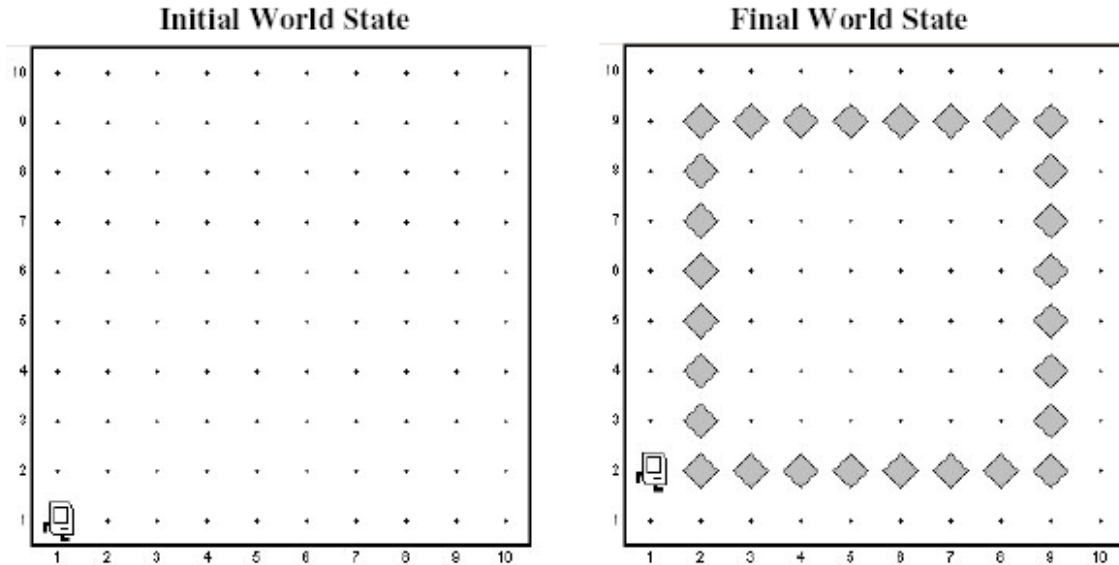
Unless otherwise indicated as part of the instructions for a specific problem, your code will not be graded on style – only on functionality. On the other hand, good style (comments, etc.) may help you to get partial credit if they help us determine what you were trying to do.

Blank pages for solutions omitted in practice exam

In an effort to save trees, the blank pages following each problem that would be provided in a regular exam for writing your solutions have been omitted from this practice exam.

Problem 1: Karel the Robot (20 points)

Write a Karel program which will create an inside border around the world. Each location that is part of the border should have one (and only one) beeper on it and the border should be inset by one square from the outer walls of the world like this:



In solving this problem, you can count on the following facts about the world:

- You may assume that the world is at least 3x3 squares. The correct solution for a 3x3 square world is to place a single beeper in the center square. However, the world is not guaranteed to be square (e.g. it could be 16x10).
- Karel starts off facing East at the corner of 1st Street and 1st Avenue with an infinite number beepers in its beeper bag, and no beepers initially in the world.
- Karel's final location or the final direction Karel is facing do not matter.
- You do not need to worry about efficiency.
- You are limited to the instructions in the Karel course reader—the only variables allowed are loop control variables used within the control section of the **for** loop.

Write your solution on the following pages (blank pages omitted to save trees).

Problem 2: Java expressions, statements, and methods (20 points)

(2a) Compute the value of each of the following Java expressions. Be sure to list a constant of the appropriate type (e.g. 7.0, not 7, for a **double**, **Strings** in quotes, etc.). If an error occurs during an evaluation, write "Error" on that line and explain briefly why the error occurs.

5.0 / 4 - 4 / 5 _____

7 < 9 - 5 && 3 % 0 == 3 _____

"B" + 8 + 4 _____

(2b) What output is printed by the following program?

```
/*
 * File: Problem2b.java
 * -----
 * This program doesn't do anything useful and exists only to test
 * your understanding of method calls and parameter passing.
 */

import acm.program.*;

public class Problem2b extends ConsoleProgram {
    public void run() {
        int num1 = 2;
        int num2 = 13;
        println("The 1st number is: " + mystery(num1, 6));
        println("The 2nd number is: " + mystery(num2 % 5, 1 + num1 * 2));
    }

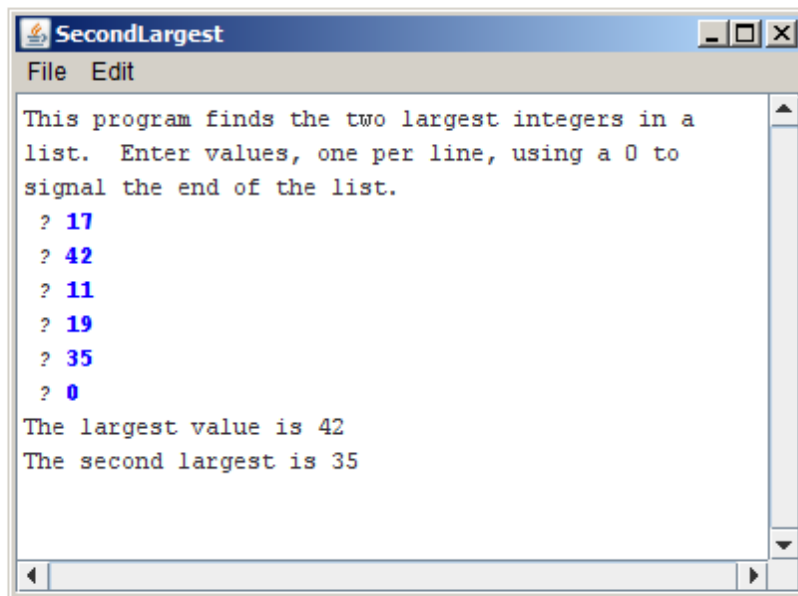
    private int mystery(int num1, int num2) {
        num1 = unknown(num1, num2);
        num2 = unknown(num2, num1);
        return num2;
    }

    private int unknown(int num1, int num2) {
        int num3 = num1 + num2;
        num2 += num3 * 2;
        return num2;
    }
}
```

Answer:

Problem 3: Console Programs (25 points)

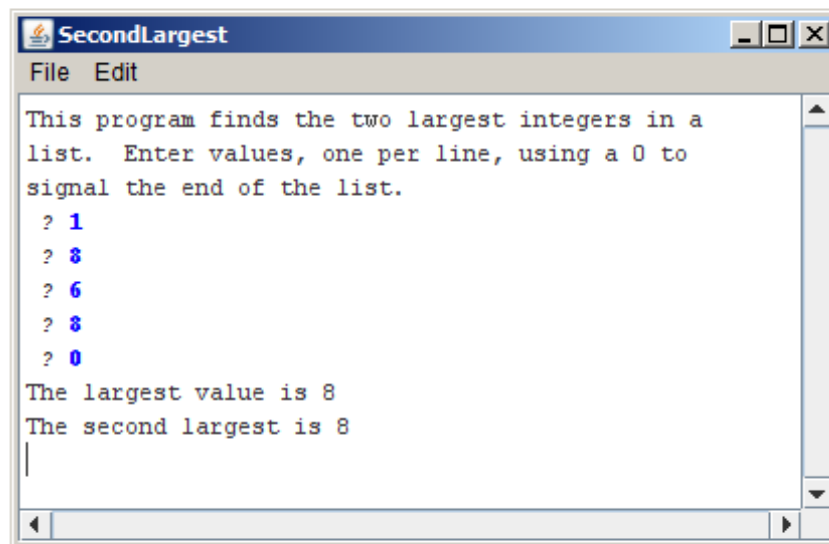
In Assignment #2, you wrote a program to find the largest and smallest temperatures (among other statistics) from a list entered by the user. For this problem, write a similar program that finds the largest and the second-largest integer entered by the user. You should use 0 as a sentinel to indicate the end of the input list. Thus, a sample run of the program might look like this:



```
SecondLargest
File Edit
This program finds the two largest integers in a
list. Enter values, one per line, using a 0 to
signal the end of the list.
? 17
? 42
? 11
? 19
? 35
? 0
The largest value is 42
The second largest is 35
```

To reduce the number of special cases, you may make the following assumptions:

- The user must enter at least two values before the sentinel.
- All input values are positive integers.
- If the largest value appears more than once, that value should be listed as both the largest and second-largest value, as shown in the following sample run:



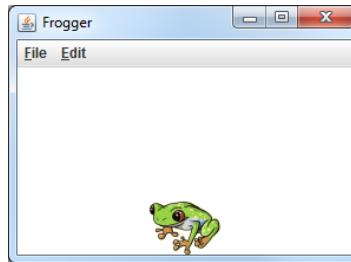
```
SecondLargest
File Edit
This program finds the two largest integers in a
list. Enter values, one per line, using a 0 to
signal the end of the list.
? 1
? 8
? 6
? 8
? 0
The largest value is 8
The second largest is 8
|
```

Write your solution on the following pages (omitted).

Problem 4: Graphics Programs (20 points)

In the arcade game Frogger, there is a frog that "hops" along the screen. A full game is beyond the scope of an exam problem, so instead let's write the code that (1) puts an image of the frog on the screen and (2) gets the frog to jump vertically when the user clicks the mouse.

Your first task in this problem is to place the frog exactly at the bottom-middle of the graphics window, as shown below. The frog is represented by an image stored in the file **frog.gif**, assumed to be in your project's **res** folder.



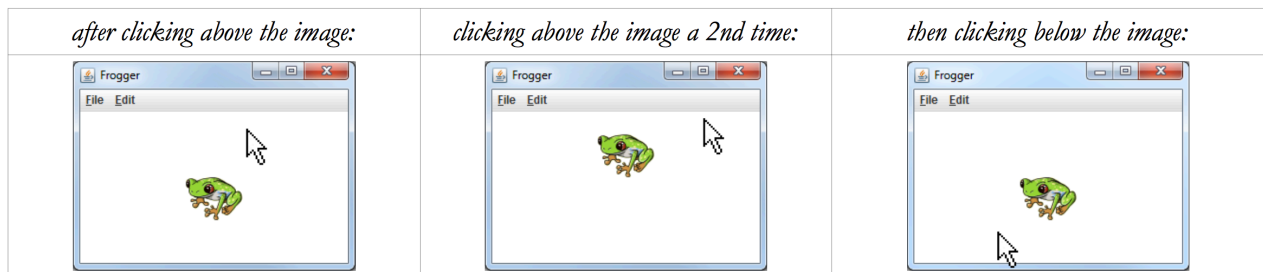
The second task in this problem is getting the frog to "hop" vertically when the user clicks the mouse. Each time the user clicks, the frog should "hop" vertically on the screen by a number of pixels exactly equal to the frog image's height. For example, if the **frog.gif** image is 60x45 (Width x Height), clicking the mouse should move the frog up or down by 45px each time.

The direction of the "hop" is dependent on the mouse click location and the frog's current position:

- If the mouse click is above the top of the frog, it should move upward on a click.
- If the mouse click is below the bottom of the frog, it should move downward on a click.
- If the mouse click is vertically within the range of pixels occupied by the frog, it should not move on a click.
- If the next hop would move the frog to an (x, y) position that is partly or entirely outside of the visible window area, the frog should not move on a click.

For this problem, your code should not make any assumptions about the window size or frog image size other than that the window size is at least as large as the frog image itself.

The following screenshots show the results of several mouse clicks above and below the frog:



Problem 5: Strings, Characters and Files (35 points)

This is a two-part problem in which you must write 2 methods.

- a) Write a method **removeDuplicates** that accepts a String parameter and returns a new string with all consecutive occurrences of the same character in the String replaced by a single occurrence of that character. The table below shows several calls to your method and their expected return values:

Method Call	Return Value
<code>removeDuplicates("bookkeeper")</code>	<code>"bokeper"</code>
<code>removeDuplicates("tresssssidder")</code>	<code>"tresider"</code>
<code>removeDuplicates("aaaAAA bbbbBBBB cccccCCC")</code>	<code>"aA bB cC"</code>
<code>removeDuplicates("banana")</code>	<code>"banana"</code>
<code>removeDuplicates("X")</code>	<code>"X"</code>
<code>removeDuplicates("")</code>	<code>""</code>

- b) Write a second method named **removeDuplicatesFromFile** that accepts a String parameter representing a filename and opens the file, reading each word in it and printing that word to the console with its duplicate letters removed. For example, if the file **myInput.txt** contains the following text:

```
tresssssidder union
iiss hiiirringg a
neeew bookkeeper !!!!
tteehhh eeend.
```

Then the call of **removeDuplicatesFromFile("myInput.txt");** should print the following console output:

```
tresider union
is hiring a

new bokeper !

teh end.
```

Notice that the duplicate characters have been removed from each word in the file, retaining the given line breaks that were present in the file originally. You may assume that the given file exists and is readable. You may also assume that each word in the file is separated by a single space.