Beijing National Day School
Department of Mathematics & Computer Science

AP Computer Science A

Section II: Free Response

NO CALCULATORS PERMITTED

English Name: ____________________

Pinyin Name: ____________________

Mr. Alwin Tareen, Spring 2020

Exam Record

Free Response ________ / 36 pts

Total: ________ / 36 pts

Grade: ________

Section II: Free Response (36 points)

- Number of questions: 4. Percent of total grade: 50%.
- **SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.**
- Assume that the classes listed in the **Java Quick Reference** have been imported where appropriate.
- Unless otherwise noted in the question, assume that parameters in method calls are not `null` and that methods are called only when their preconditions are satisfied.
- In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods may not receive full credit.

`CookieOrder` question(2010 AP Computer Science A Free Response)

(9^{pts}) **1.** An organization raises money by selling boxes of cookies. A cookie order specifies the variety of cookie and the number of boxes ordered. The declaration of the `CookieOrder` class is shown below.

```java
public class CookieOrder
{
    /** Constructs a new CookieOrder object.
     */
    public CookieOrder(String variety, int numBoxes)
    { /* implementation not shown */ }

    /** @return the variety of cookie being ordered.
     */
    public String getVariety()
    { /* implementation not shown */ }

    /** @return the number of boxes being ordered.
     */
    public int getNumBoxes()
    { /* implementation not shown */ }

    // There may be instance variables, constructors, and methods that are not shown.
}
```

The `MasterOrder` class maintains a list of the cookies to be purchased. The declaration of the `MasterOrder` class is shown below.

```
public class MasterOrder
{
    /** The list of all cookie orders.
     */
    private ArrayList<CookieOrder> orders;

    /** Constructs a new MasterOrder object.
     */
    public MasterOrder()
    { orders = new ArrayList<CookieOrder>(); }

    /** Adds theOrder to the master order.
     * @param theOrder the cookie order to add to the master order.
     */
    public void addOrder(CookieOrder theOrder)
    { orders.add(theOrder); }

    /** @return the sum of the number of boxes of all of the cookie orders.
     */
    public int getTotalBoxes()
    { /* to be implemented in part (a) */ }

    /** Removes all cookie orders from the master order that have the same variety of
     * cookie as cookieVar and returns the total number of boxes that were removed.
     * @param cookieVar the variety of cookies to remove from the master order.
     * @return the total number of boxes of cookieVar in the cookie orders removed.
     */
    public int removeVariety(String cookieVar)
    { /* to be implemented in part (b) */ }

    // There may be instance variables, constructors, and methods that are not shown.
}
```

(a) (3 pts) The `getTotalBoxes` method computes and returns the sum of the number of boxes of all cookie orders. If there are no cookie orders in the master order, the method returns `0`.

Complete method `getTotalBoxes` below.

```
/** @return the sum of the number of boxes of all of the cookie orders.
 */
public int getTotalBoxes()
```

(b) The `removeVariety` method updates the master order by removing all of the cookie orders in which the variety of cookie matches the parameter `cookieVar`. The master order may contain zero or more cookie orders with the same variety as `cookieVar`. The method returns the total number of boxes removed from the master order.

For example, consider the following code segment:

```
MasterOrder goodies = new MasterOrder();
goodies.addOrder(new CookieOrder("Chocolate Chip", 1));
goodies.addOrder(new CookieOrder("Shortbread", 5));
goodies.addOrder(new CookieOrder("Macaroon", 2));
goodies.addOrder(new CookieOrder("Chocolate Chip", 3));
```

After the code segment has executed, the contents of the master order are as shown in the following table.

"Chocolate Chip" 1	"Shortbread" 5	"Macaroon" 2	"Chocolate Chip" 3

The method call `goodies.removeVariety("Chocolate Chip")` returns `4` because there were two Chocolate Chip cookie orders totalling `4` boxes. The master order is modified as shown below.

"Shortbread" 5	"Macaroon" 2

The method call `goodies.removeVariety("Brownie")` returns `0` and does **not** change the master order.

Write your solution on the next page.

(b) (6 pts) Complete method `removeVariety` below.

```
/** Removes all cookie orders from the master order that have the same variety of
 * cookie as cookieVar and returns the total number of boxes that were removed.
 * @param cookieVar the variety of cookies to remove from the master order.
 * @return the total number of boxes of cookieVar in the cookie orders removed.
 */
public int removeVariety(String cookieVar)
```

`APLine` Question(2010 AP CompSci Free Response)

(9pts) **2.** An `APLine` is a line defined by the equation $ax + by + c = 0$, where a is not equal to zero, b is not equal to zero, and a, b, and c are all integers. The slope of an `APLine` is defined to be the `double` value $-a/b$.

A point(represented by integers x and y) is on an `APLine` if the equation of the `APLine` is satisfied when those x and y values are substituted into the equation. That is, a point represented by x and y is on the line if $ax + by + c$ is equal to 0. Examples of two `APLine` equations are shown in the following table.

Equation	Slope$(-a/b)$	Is point $(5, 2)$ on the line?
$5x + 4y - 17 = 0$	$-5/4 = -1.25$	Yes, because $5(5) + 4(-2) + (-17) = 0$
$-25x + 40y + 30 = 0$	$25/40 = 0.625$	No, because $-25(5) + 40(-2) + 30 \neq 0$

Assume that the following code segment appears in a class other than `APLine`. The code segment shows an example of using the `APLine` class to represent the two equations shown in the table.

```
APLine line1 = new APLine(5, 4, −17);
double slope1 = line1.getSlope(); // slope1 is assigned −1.25
boolean onLine1 = line1.isOnLine(5, −2); // true

APLine line2 = new APLine(−25, 40, 30);
double slope2 = line2.getSlope(); // slope2 is assigned 0.625
boolean onLine2 = line2.isOnLine(5, −2); // false
```

Write the `APLine` class. Your class must produce the indicated results when invoked by the code segment given above. You may ignore any issues related to integer overflow. Your implementation must include:

(a) (1 pt) The declaration of the private instance variables `a`, `b` and `c`.

(b) (2 pts) A constructor that has three integer parameters that represent `a`, `b`, and `c`, in that order. You may assume that the values of the parameters representing `a` and `b` are not zero.

(c) (3 pts) A method `getSlope()` that calculates and returns the slope of the line.

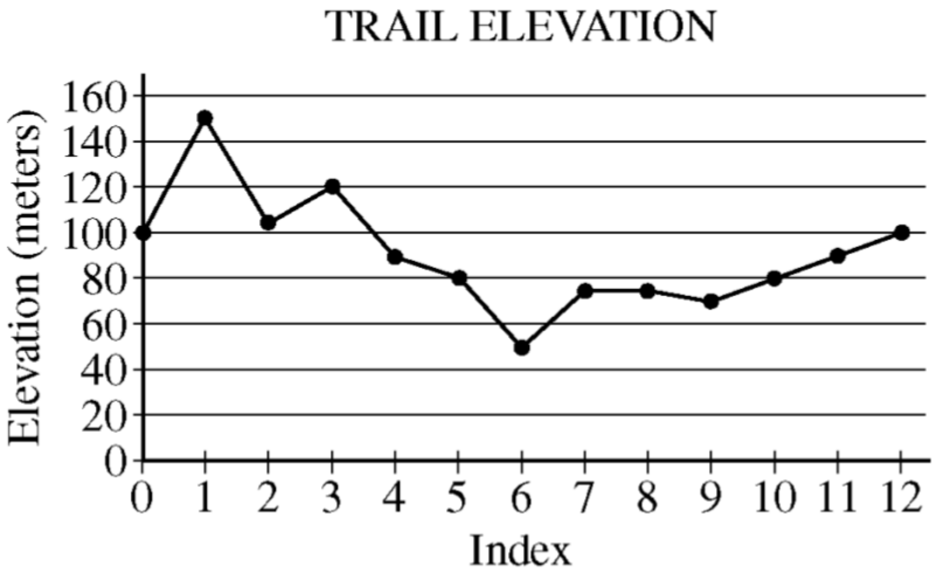
(d) (3 pts) A method `isOnLine(int x, int y)` that returns `true` if the point represented by its two parameters (`x` and `y`, in that order) is on the `APLine`, and returns `false` otherwise.

Write your solution on the next page.

Complete `APLine.java` in the space below.

`Trail` question(2010 AP CompSci Free Response)

(9pts) **3.** A hiking trail has elevation markers posted at regular intervals along the trail. Elevation information about a trail can be stored in an array, where each element in the array represents the elevation at a marker. The elevation at the first marker will be stored at array index 0, the elevation at the second marker will be stored at array index 1, and so forth. Elevations between markers are ignored in this question. The graph below shows an example of trail elevations.



The table below contains the data represented in the graph.

Trail Elevation (meters)

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Elevation	100	150	105	120	90	80	50	75	75	70	80	90	100

The declaration of the `Trail` class is shown below. You will write two unrelated methods of the `Trail` class.

```
public class Trail
{
    /** Representation of the trail. The number of markers on the trail is markers.length.
    */
    private int[] markers;

    /** Determines if a trail segment is level. A trail segment is defined by a starting marker,
    * an ending marker, and all markers between those two markers.
    *
    * A trail segment is level if it has a difference between the max elevation and min
    * elevation that is less than or equal to 10m.
    *
    * @param start The index of the starting marker.
    * @param end The index of the ending marker.
    *
    * Precondition: 0 <= start < end <= markers.length - 1
    * @return true If the difference between the maximum and minimum elevation on
    * this segment of the trail is less than or equal to 10 meters; false otherwise.
    */
    public boolean isLevelTrailSegment(int start, int end)
    { /* to be implemented in part (a) */ }

    /** Determines if this trail is rated difficult. A trail is rated by counting the number of
    * changes in elevation that are at least 30 meters (up or down) between two consecutive
    * markers. A trail with 3 or more such changes is rated difficult.
    * @return true if the trail is rated difficult; false otherwise.
    */
    public boolean isDifficult()
    { /* to be implemented in part (b) */ }

    // There may be instance variables, constructors, and methods that are not shown.
}
```

(a) (5 pts) Write the `Trail` method `isLevelTrailSegment`.

- A trail segment is defined by a starting marker, an ending marker, and all markers between those two markers. The parameters of the method are the index of the starting marker and the index of the ending marker. The method will return `true` if the difference between the maximum elevation and the minimum elevation in the trail segment is less than or equal to 10 meters.
- For the trail shown at the beginning of the question, the trail segment starting at marker 7 and ending at marker 10 has elevations ranging between 70 and 80 meters. Because the difference between 80 and 70 is equal to 10, the trail segment is considered level, so the method will return `true`.
- The trail segment starting at marker 2 and ending at marker 12 has elevations ranging between 50 and 120 meters. Because the difference between 120 and 50 is greater than 10, this trail segment is not considered level, so the method will return `false`.

Complete method `isLevelTrailSegment` below.

```
/** Determines if a trail segment is level. A trail segment is defined by a starting marker,
 * an ending marker, and all markers between those two markers.
 *
 * A trail segment is level if it has a difference between the max elevation and min
 * elevation that is less than or equal to 10m.
 *
 * @param start The index of the starting marker.
 * @param end The index of the ending marker.
 *
 * Precondition: 0 <= start < end <= markers.length - 1
 * @return true If the difference between the maximum and minimum elevation on
 * this segment of the trail is less than or equal to 10 meters; false otherwise.
 */
public boolean isLevelTrailSegment(int start, int end)
```

(b) (4 pts) Write the `Trail` method `isDifficult`.

- A trail is rated by counting the number of changes in elevation that are at least 30 meters (up or down) between two consecutive markers. A trail with 3 or more such changes is rated difficult. The following table shows trail elevation data and the elevation changes between consecutive trail markers.
- If the trail is determined to be difficult, the method will return `true`. Otherwise, if the trail is not difficult, the method will return `false`.

Trail Elevation (meters)

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Elevation	100	150	105	120	90	80	50	75	75	70	80	90	100

	\ /	\ /	\ /	\ /	\ /	\ /	\ /	\ /	\ /	\ /	\ /	\ /
Elevation change	50	-45	15	-30	-10	-30	25	0	-5	10	10	10

- This trail is rated difficult because it has 4 changes in elevation that are 30 meters or more (between markers 0 and 1, between markers 1 and 2, between markers 3 and 4, and between markers 5 and 6). Therefore, the method will return `true` in this case.

Write your solution on the next page.

Complete method `isDifficult` below.

```
/** Determines if this trail is rated difficult. A trail is rated by counting the number of
 * changes in elevation that are at least 30 meters (up or down) between two consecutive
 * markers. A trail with 3 or more such changes is rated difficult.
 * @return true if the trail is rated difficult; false otherwise.
 */
public boolean isDifficult()
```

GrayImage question(2012 AP Computer Science A Free Response)

(9pts) **4.** A grayscale image is represented by a 2-dimensional rectangular array of pixels(picture elements). A pixel is an integer value that represents a shade of gray. In this question, pixel values can be in the range from 0 through 255, inclusive. A black pixel is represented by 0, and a white pixel is represented by 255.

The declaration of the GrayImage class is shown below. You will write two unrelated methods of the GrayImage class.

```
public class GrayImage
{
    public static final int BLACK = 0;
    public static final int WHITE = 255;

    /** The 2-dimensional representation of this image. Guaranteed not to be null.
     * All values in the array are within the range [BLACK, WHITE], inclusive.
     */
    private int[] [] pixelValues;

    /** @return the total number of white pixels in this image.
     * Postcondition: this image has not been changed.
     */
    public int countWhitePixels()
    { /* to be implemented in part (a) */ }

    /** Processes this image in row-major order and decreases the value of each pixel at
     * position(row, col) by the value of the pixel at position(row+2, col+2) if it exists.
     * Resulting values that would be less than BLACK are replaced by BLACK.
     * Pixels for which there is no pixel at position(row+2, col+2) are unchanged.
     */
    public void processImage()
    { /* to be implemented in part (b) */ }
}
```

(a) Write the method countWhitePixels that returns the number of pixels in the image that contain the value WHITE. For example, assume that pixelValues contains the following image:

	0	1	2	3	4
0	**255**	184	178	84	129
1	84	**255**	**255**	130	84
2	78	**255**	0	0	78
3	84	130	**255**	130	84

A call to the countWhitePixels method would return 5 because there are 5 entries(shown in boldface) that have the value WHITE.

(a) (4 pts) Complete method `countWhitePixels` below.

```
/** @return the total number of white pixels in this image.
 * Postcondition: this image has not been changed.
 */
public int countWhitePixels()
```

(b) Write the method `processImage` that modifies the image by changing the values in the instance variable `pixelValues` according to the following description. The pixels in the image are processed one at a time in **row-major order**.

Row-major order processes the first row in the array from left-to-right, and then processes the second row from left-to-right, continuing until all rows are processed from left-to-right. The first index of `pixelValues` represents the row number, and the second index represents the column number.

The pixel value at position(row, col) is decreased by the value at position (row+2, col+2) if such a position exists. If the result of the subtraction is less than the value `BLACK`, the pixel is assigned the value of `BLACK`. The values of the pixels for which there is no pixel at position(row+2, col+2) remain unchanged. You may assume that all the original values in the array are within the range `[BLACK, WHITE]`, inclusive.

The following diagram shows the contents of the instance variable `pixelValues` before and after a call to `processImage`. The values shown in boldface represent the pixels that could be modified in a grayscale image with 4 rows and 5 columns.

Before Call to `processImage`

	0	1	2	3	4
0	**221**	**184**	**178**	84	135
1	**84**	**255**	**255**	130	84
2	78	255	0	0	78
3	84	130	255	130	84

After Call to `processImage`

	0	1	2	3	4
0	**221**	**184**	**100**	84	135
1	**0**	**125**	**171**	130	84
2	78	255	0	0	78
3	84	130	255	130	84

Information repeated from the beginning of the question:

public class GrayImage

public static final int BLACK = 0
public static final int WHITE = 255
private int[][] pixelValues
public int countWhitePixels()
public void processImage()

Write your solution on the next page.

(b) (5 pts) Complete method `processImage` below.

```
/** Processes this image in row−major order and decreases the value of each pixel at
 * position(row, col) by the value of the pixel at position(row+2, col+2) if it exists.
 * Resulting values that would be less than BLACK are replaced by BLACK.
 * Pixels for which there is no pixel at position(row+2, col+2) are unchanged.
 */
public void processImage()
```

STOP
END OF EXAM

This page is left intentionally blank.