AP® Computer Science A
2013 Free-Response Questions

About the College Board

The College Board is a mission-driven not-for-profit organization that connects students to college success and opportunity. Founded in 1900, the College Board was created to expand access to higher education. Today, the membership association is made up of more than 6,000 of the world's leading educational institutions and is dedicated to promoting excellence and equity in education. Each year, the College Board helps more than seven million students prepare for a successful transition to college through programs and services in college readiness and college success — including the SAT® and the Advanced Placement Program®. The organization also serves the education community through research and advocacy on behalf of students, educators, and schools.

Visit the College Board on the Web: www.collegeboard.org.
AP Central is the official online home for the AP Program: apcentral.collegeboard.org.

COMPUTER SCIENCE A
SECTION II
Time—1 hour and 45 minutes
Number of questions—4
Percent of total score—50

Directions: SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.

Notes:
- Assume that the classes listed in the appendices 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.

1. A music Web site keeps track of downloaded music. For each download, the site uses a `DownloadInfo` object to store a song's title and the number of times it has been downloaded. A partial declaration for the `DownloadInfo` class is shown below.

```
public class DownloadInfo
{
  /** Creates a new instance with the given unique title and sets the
   *  number of times downloaded to 1.
   *  @param title the unique title of the downloaded song
   */
  public DownloadInfo(String title)
  {  /* implementation not shown */  }


  /** @return the title */
  public String getTitle()
  {  /* implementation not shown */  }


  /** Increment the number times downloaded by 1 */
  public void incrementTimesDownloaded()
  {  /* implementation not shown */  }

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

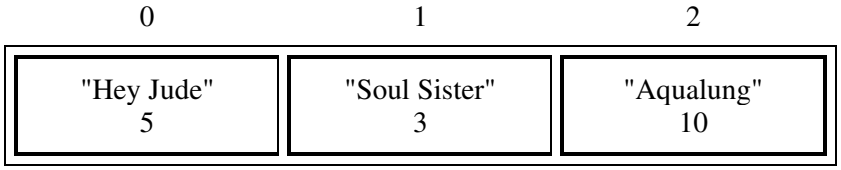
The list of downloaded information is stored in a `MusicDownloads` object. A partial declaration for the `MusicDownloads` class is shown below.

```
public class MusicDownloads
{
  /** The list of downloaded information.
   *  Guaranteed not to be null and not to contain duplicate titles.
   */
  private List<DownloadInfo> downloadList;


  /** Creates the list of downloaded information. */
  public MusicDownloads()
  {  downloadList = new ArrayList<DownloadInfo>();  }


  /** Returns a reference to the DownloadInfo object with the requested title if it exists.
   *  @param title the requested title
   *  @return a reference to the DownloadInfo object with the
   *          title that matches the parameter title if it exists in the list;
   *          null otherwise.
   *  Postcondition:
   *    - no changes were made to downloadList.
   */
  public DownloadInfo getDownloadInfo(String title)
  {  /* to be implemented in part (a) */  }


  /** Updates downloadList with information from titles.
   *  @param titles a list of song titles
   *  Postcondition:
   *    - there are no duplicate titles in downloadList.
   *    - no entries were removed from downloadList.
   *    - all songs in titles are represented in downloadList.
   *    - for each existing entry in downloadList, the download count is increased by
   *        the number of times its title appeared in titles.
   *    - the order of the existing entries in downloadList is not changed.
   *    - the first time an object with a title from titles is added to downloadList, it
   *        is added to the end of the list.
   *    - new entries in downloadList appear in the same order
   *        in which they first appear in titles.
   *    - for each new entry in downloadList, the download count is equal to
   *        the number of times its title appeared in titles.
   */
  public void updateDownloads(List<String> titles)
  {  /* to be implemented in part (b) */  }

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

Part (a) begins on page 4.

(a) Write the `MusicDownloads` method `getDownloadInfo`, which returns a reference to a `DownloadInfo` object if an object with a title that matches the parameter `title` exists in the `downloadList`. If no song in `downloadList` has a title that matches the parameter `title`, the method returns `null`.

For example, suppose variable `webMusicA` refers to an instance of `MusicDownloads` and that the table below represents the contents of `downloadList`. The list contains three `DownloadInfo` objects. The object at position 0 has a title of "Hey Jude" and a download count of 5. The object at position 1 has a title of "Soul Sister" and a download count of 3. The object at position 2 has a title of "Aqualung" and a download count of 10.



The call `webMusicA.getDownloadInfo("Aqualung")` returns a reference to the object in position 2 of the list.

The call `webMusicA.getDownloadInfo("Happy Birthday")` returns `null` because there are no `DownloadInfo` objects with that title in the list.

Class information repeated from the beginning of the question

```
public class DownloadInfo

public DownloadInfo(String title)
public String getTitle()
public void incrementTimesDownloaded()

public class MusicDownloads

private List<DownloadInfo> downloadList
public DownloadInfo getDownloadInfo(String title)
public void updateDownloads(List<String> titles)
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

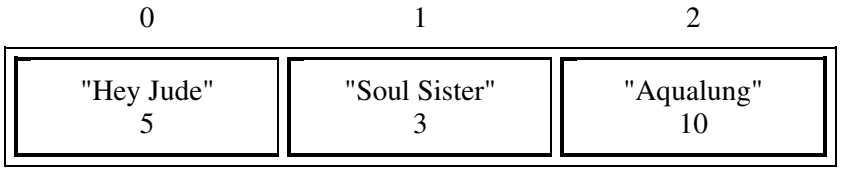
Complete method `getDownloadInfo` below.

```
/** Returns a reference to the DownloadInfo object with the requested title if it exists.
 *  @param title the requested title
 *  @return a reference to the DownloadInfo object with the
 *          title that matches the parameter title if it exists in the list;
 *          null otherwise.
 *  Postcondition:
 *    - no changes were made to downloadList.
 */
public DownloadInfo getDownloadInfo(String title)
```

Part (b) begins on page 6.

(b) Write the `MusicDownloads` method `updateDownloads`, which takes a list of song titles as a parameter. For each title in the list, the method updates `downloadList`, either by incrementing the download count if a `DownloadInfo` object with the same title exists, or by adding a new `DownloadInfo` object with that title and a download count of 1 to the end of the list. When a new `DownloadInfo` object is added to the end of the list, the order of the already existing entries in `downloadList` remains unchanged.

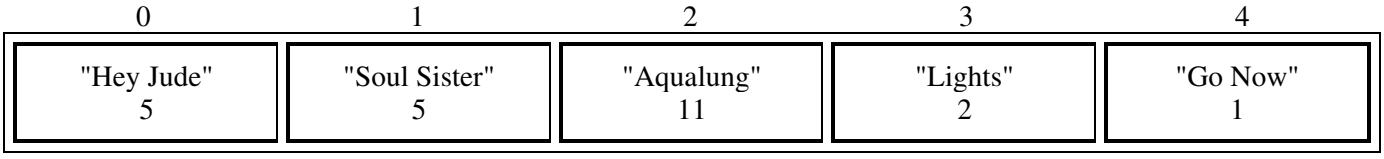
For example, suppose variable `webMusicB` refers to an instance of `MusicDownloads` and that the table below represents the contents of the instance variable `downloadList`.



Assume that the variable `List<String> songTitles` has been defined and contains the following entries.

```
{"Lights", "Aqualung", "Soul Sister", "Go Now", "Lights", "Soul Sister"}
```

The call `webMusicB.updateDownloads(songTitles)` results in the following `downloadList` with incremented download counts for the objects with titles of "Soul Sister" and "Aqualung". It also has a new `DownloadInfo` object with a title of "Lights" and a download count of 2, and another `DownloadInfo` object with a title of "Go Now" and a download count of 1. The order of the already existing entries remains unchanged.



Class information repeated from the beginning of the question

```
public class DownloadInfo

public DownloadInfo(String title)
public String getTitle()
public void incrementTimesDownloaded()

public class MusicDownloads

private List<DownloadInfo> downloadList
public DownloadInfo getDownloadInfo(String title)
public void updateDownloads(List<String> titles)
```

In writing your solution, you must use the `getDownloadInfo` method. Assume that `getDownloadInfo` works as specified, regardless of what you wrote for part (a).

Complete method `updateDownloads` below.

```
/** Updates downloadList with information from titles.
 *  @param titles a list of song titles
 *  Postcondition:
 *    – there are no duplicate titles in downloadList.
 *    – no entries were removed from downloadList.
 *    – all songs in titles are represented in downloadList.
 *    – for each existing entry in downloadList, the download count is increased by
 *        the number of times its title appeared in titles.
 *    – the order of the existing entries in downloadList is not changed.
 *    – the first time an object with a title from titles is added to downloadList, it
 *        is added to the end of the list.
 *    – new entries in downloadList appear in the same order
 *        in which they first appear in titles.
 *    – for each new entry in downloadList, the download count is equal to
 *        the number of times its title appeared in titles.
 */
public void updateDownloads(List<String> titles )
```

2. A multiplayer game called Token Pass has the following rules.

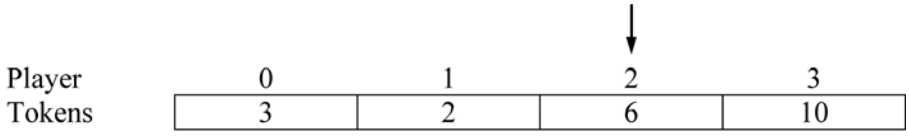
Each player begins with a random number of tokens (at least 1, but no more than 10) that are placed on a linear game board. There is one position on the game board for each player. After the game board has been filled, a player is randomly chosen to begin the game. Each position on the board is numbered, starting with 0.

The following rules apply for a player's turn.

- The tokens are collected and removed from the game board at that player's position.
- The collected tokens are distributed one at a time, to each player, beginning with the next player in order of increasing position.
- If there are still tokens to distribute after the player at the highest position gets a token, the next token will be distributed to the player at position 0.
- The distribution of tokens continues until there are no more tokens to distribute.

The Token Pass game board is represented by an array of integers. The indexes of the array represent the player positions on the game board, and the corresponding values in the array represent the number of tokens that each player has. The following example illustrates one player's turn.

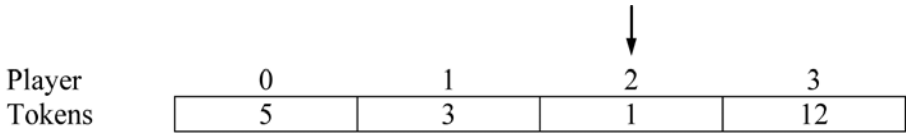
Example
The following represents a game with 4 players. The player at position 2 was chosen to go first.



The tokens at position 2 are collected and distributed as follows.

1st token - to position 3 (The highest position is reached, so the next token goes to position 0.)
2nd token - to position 0
3rd token - to position 1
4th token - to position 2
5th token - to position 3 (The highest position is reached, so the next token goes to position 0.)
6th token - to position 0

After player 2's turn, the values in the array will be as follows.



The Token Pass game is represented by the `TokenPass` class.

```
public class TokenPass
{
  private int[] board;
  private int currentPlayer;


  /** Creates the board array to be of size playerCount and fills it with
   *  random integer values from 1 to 10, inclusive. Initializes currentPlayer to a
   *  random integer value in the range between 0 and playerCount-1, inclusive.
   *  @param playerCount the number of players
   */
  public TokenPass(int playerCount)
  {  /* to be implemented in part (a) */  }


  /** Distributes the tokens from the current player's position one at a time to each player in
   *  the game. Distribution begins with the next position and continues until all the tokens
   *  have been distributed. If there are still tokens to distribute when the player at the
   *  highest position is reached, the next token will be distributed to the player at position 0.
   *  Precondition: the current player has at least one token.
   *  Postcondition: the current player has not changed.
   */
  public void distributeCurrentPlayerTokens()
  {  /* to be implemented in part (b) */  }

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

(a) Write the constructor for the `TokenPass` class. The parameter `playerCount` represents the number of players in the game. The constructor should create the `board` array to contain `playerCount` elements and fill the array with random numbers between 1 and 10, inclusive. The constructor should also initialize the instance variable `currentPlayer` to a random number between 0 and `playerCount-1`, inclusive.

Complete the `TokenPass` constructor below.

```
/** Creates the board array to be of size playerCount and fills it with
 *  random integer values from 1 to 10, inclusive. Initializes currentPlayer to a
 *  random integer value in the range between 0 and playerCount-1, inclusive.
 *  @param playerCount the number of players
 */
public TokenPass(int playerCount)
```

Part (b) begins on page 11.

(b) Write the `distributeCurrentPlayerTokens` method.

The tokens are collected and removed from the game board at the current player's position. These tokens are distributed, one at a time, to each player, beginning with the next higher position, until there are no more tokens to distribute.

Class information repeated from the beginning of the question

```
public class TokenPass

private int[] board
private int currentPlayer
public TokenPass(int playerCount)
public void distributeCurrentPlayerTokens()
```

Complete method `distributeCurrentPlayerTokens` below.

```
/** Distributes the tokens from the current player's position one at a time to each player in
 *  the game. Distribution begins with the next position and continues until all the tokens
 *  have been distributed. If there are still tokens to distribute when the player at the
 *  highest position is reached, the next token will be distributed to the player at position 0.
 *  Precondition: the current player has at least one token.
 *  Postcondition: the current player has not changed.
 */
public void distributeCurrentPlayerTokens()
```

3. This question involves reasoning about the GridWorld case study. Reference materials are provided in the appendixes. In part (a) you will write a method to return an array list of all empty locations in a given grid. In part (b) you will write the class for a new type of `Critter`.

   (a) The `GridWorldUtilities` class contains `static` methods. A partial declaration of the `GridWorldUtilities` class is shown below.

```
public class GridWorldUtilities
{
  /** Gets all the locations in grid that do not contain objects.
   *  @param grid a reference to a BoundedGrid object
   *  @return an array list (possibly empty) of empty locations in grid.
   *          The size of the returned list is 0 if there are no empty locations in grid.
   *          Each empty location in grid should appear exactly once in the returned list.
   */
  public static ArrayList<Location> getEmptyLocations(Grid<Actor> grid)
  {  /* to be implemented in part (a) */  }

  // There may be instance variables that are not shown.
}
```

   Write the `GridWorldUtilities` method `getEmptyLocations`. If there are no empty locations in `grid`, the method returns an empty array list. Otherwise, it returns an array list of all empty locations in `grid`. Each empty location should appear exactly once in the array list.

WRITE YOUR SOLUTION ON THE NEXT PAGE.

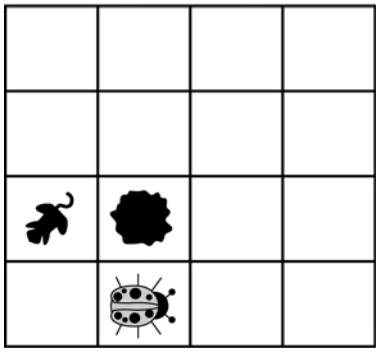
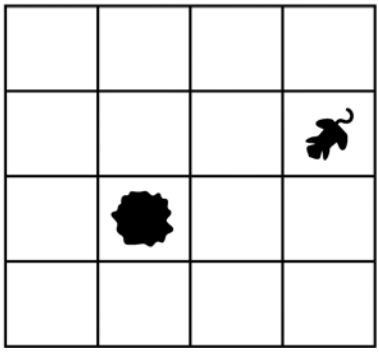
Complete method `getEmptyLocations` below.

```
/** Gets all the locations in grid that do not contain objects.
 *  @param grid a reference to a BoundedGrid object
 *  @return an array list (possibly empty) of empty locations in grid.
 *          The size of the returned list is 0 if there are no empty locations in grid.
 *          Each empty location in grid should appear exactly once in the returned list.
 */
public static ArrayList<Location> getEmptyLocations(Grid<Actor> grid)
```

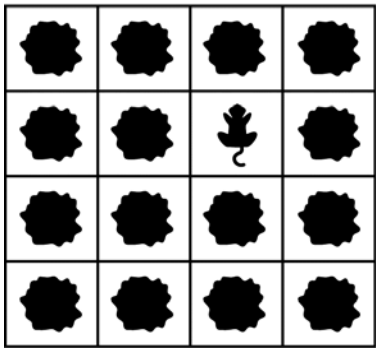
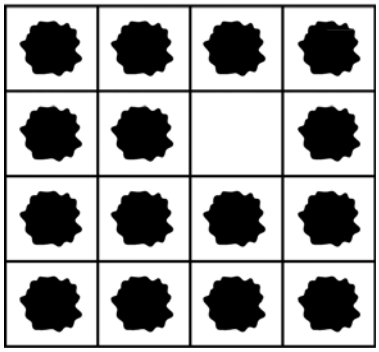
Part (b) begins on page 14.

(b) A `JumpingCritter` acts like a `Critter`, except that it moves by jumping to a randomly selected empty location in its grid. If there are no empty locations, the `JumpingCritter` removes itself from the grid.

The following diagram shows an example of a jumping critter that is able to move to an empty location. Example World #1 is shown below on the left. After the jumping critter at location (2, 0) acts, the world shown below on the right is one possible result.

EXAMPLE WORLD #1	POSSIBLE WORLD AFTER ACT
	
A jumping critter is in location (2, 0).	The jumping critter has eaten the bug that was in location (3, 1) and has moved to location (1, 3).

Example World #2 is shown below on the left. After the jumping critter at location (1, 2) acts, the world shown below on the right is the result.

EXAMPLE WORLD #2	WORLD AFTER ACT
	
A jumping critter is in location (1, 2).	The jumping critter removed itself from the grid because no empty locations were available.

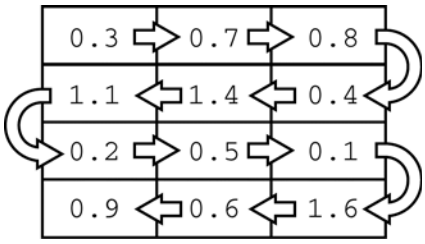
Class information repeated from the beginning of the question

```
public class GridWorldUtilities

public static ArrayList<Location> getEmptyLocations(Grid<Actor> grid)
```

Assume that the `GridWorldUtilities getEmptyLocations` method works as specified, regardless of what you wrote in part (a). Solutions that reimplement the functionality of this method will not receive full credit.

Write the complete `JumpingCritter` class. Do NOT override the `act` method. Remember that your design must not violate the postconditions of the methods of the `Critter` class.

4. A telescope scans a rectangular area of the night sky and collects the data into a 1-dimensional array. Each data value scanned is a number representing the amount of light detected by the telescope. The telescope scans back and forth across the sky (alternating between left to right and right to left) in the pattern indicated below by the arrows. The back-and-forth ordering of the values received from the scan is called *telescope order*.



The telescope records the data in telescope order into a 1-dimensional array of `double` values. This 1-dimensional array of information received from a single scan will be transferred into a 2-dimensional array, which reconstructs the original view of the rectangular area of the sky. This 2-dimensional array is part of the `SkyView` class, shown below. In this question you will write a constructor and a method for this class.

```
public class SkyView
{
  /** A rectangular array that holds the data representing a rectangular area of the sky. */
  private double[][] view;

  /** Constructs a SkyView object from a 1-dimensional array of scan data.
   *  @param numRows the number of rows represented in the view
   *         Precondition: numRows > 0
   *  @param numCols the number of columns represented in the view
   *         Precondition: numCols > 0
   *  @param scanned the scan data received from the telescope, stored in telescope order
   *         Precondition: scanned.length == numRows * numCols
   *  Postcondition: view has been created as a rectangular 2-dimensional array
   *                 with numRows rows and numCols columns and the values in
   *                 scanned have been copied to view and are ordered as
   *                 in the original rectangular area of sky.
   */
  public SkyView(int numRows, int numCols, double[] scanned)
  {  /* to be implemented in part (a) */  }

  /** Returns the average of the values in a rectangular section of view.
   *  @param startRow the first row index of the section
   *  @param endRow the last row index of the section
   *  @param startCol the first column index of the section
   *  @param endCol the last column index of the section
   *  Precondition: 0 <= startRow <= endRow < view.length
   *  Precondition: 0 <= startCol <= endCol < view[0].length
   *  @return the average of the values in the specified section of view
   */
  public double getAverage(int startRow, int endRow,
                           int startCol, int endCol)
  {  /* to be implemented in part (b) */  }

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

(a) Write the constructor for the `SkyView` class. The constructor initializes the `view` instance variable to a 2-dimensional array with `numRows` rows and `numCols` columns. The information from `scanned`, which is stored in the telescope order, is copied into `view` to reconstruct the sky view as originally seen by the telescope. The information in `scanned` must be rearranged as it is stored into `view` so that the sky view is oriented properly.

For example, suppose `scanned` contains values, as shown in the following array.

	0	1	2	3	4	5	6	7	8	9	10	11
scanned	0.3	0.7	0.8	0.4	1.4	1.1	0.2	0.5	0.1	1.6	0.6	0.9

Using the `scanned` array above, a `SkyView` object created with `new SkyView(4, 3, scanned)`, would have `view` initialized with the following values.

view	0	1	2
0	0.3	0.7	0.8
1	1.1	1.4	0.4
2	0.2	0.5	0.1
3	0.9	0.6	1.6

For another example, suppose `scanned` contains the following values.

	0	1	2	3	4	5
scanned	0.3	0.7	0.8	0.4	1.4	1.1

A `SkyView` object created with `new SkyView(3, 2, scanned)`, would have `view` initialized with the following values.

view	0	1
0	0.3	0.7
1	0.4	0.8
2	1.4	1.1

Complete the `SkyView` constructor below.

```
/** Constructs a SkyView object from a 1-dimensional array of scan data.
 *  @param numRows the number of rows represented in the view
 *         Precondition: numRows > 0
 *  @param numCols the number of columns represented in the view
 *         Precondition: numCols > 0
 *  @param scanned the scan data received from the telescope, stored in telescope order
 *         Precondition: scanned.length == numRows * numCols
 *  Postcondition: view has been created as a rectangular 2-dimensional array
 *                 with numRows rows and numCols columns and the values in
 *                 scanned have been copied to view and are ordered as
 *                 in the original rectangular area of sky.
 */
public SkyView(int numRows, int numCols, double[] scanned)
```

Part (b) begins on page 19.

(b) Write the `SkyView` method `getAverage`, which returns the average of the elements of the section of `view` with row indexes from `startRow` through `endRow`, inclusive, and column indexes from `startCol` through `endCol`, inclusive.

For example, if `nightSky` is a `SkyView` object where `view` contains the values shown below, the call `nightSky.getAverage(1, 2, 0, 1)` should return `0.8`. (The average is `(1.1 + 1.4 + 0.2 + 0.5) / 4`, which equals `0.8`). The section being averaged is indicated by the dark outline in the table below.

view	0	1	2
0	0.3	0.7	0.8
1	1.1	1.4	0.4
2	0.2	0.5	0.1
3	0.9	0.6	1.6

Class information repeated from the beginning of the question

```
public class SkyView

private double[][] view
public SkyView(int numRows, int numCols, double[] scanned)
public double getAverage(int startRow, int endRow,
                         int startCol, int endCol)
```

WRITE YOUR SOLUTION ON THE NEXT PAGE.

Complete method `getAverage` below.

```
/** Returns the average of the values in a rectangular section of view.
 *  @param startRow the first row index of the section
 *  @param endRow the last row index of the section
 *  @param startCol the first column index of the section
 *  @param endCol the last column index of the section
 *  Precondition: 0 <= startRow <= endRow < view.length
 *  Precondition: 0 <= startCol <= endCol < view[0].length
 *  @return the average of the values in the specified section of view
 */
public double getAverage(int startRow, int endRow,
                         int startCol, int endCol)
```

STOP

END OF EXAM