2025

AP® Computer Science A

Free-Response Questions

COMPUTER SCIENCE A

SECTION II

TIME – 1 HOUR AND 30 MINUTES

Directions:

Section II has 4 free-response questions and lasts 1 hour and 30 minutes.

All program segments must be written in Java. Show all your work. Credit for partial solutions will be given.

Java Quick Reference information can be accessed in this application and is available throughout the exam.

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 will not receive full credit.

You may use the available paper for scratch work, but credit will only be given for responses entered in this application. Text you enter as an annotation will **not** be included as part of your answer.

You can go back and forth between questions in this section until time expires. The clock will turn red when 5 minutes remain—**the proctor will not give you any time updates or warnings.**

Note: This exam was originally administered digitally. It is presented here in a format optimized for teacher and student use in the classroom.

---

1. This question involves dog walkers who are paid through a dog-walking company to take dogs for one-hour walks. A dog-walking company has a varying number of dogs that need to be walked during each hour of the day. A dog-walking company is represented by the following `DogWalkCompany` class.

```
public class DogWalkCompany
{
  /**
   * Returns the number of dogs, always greater than 0, that are available
   * for a walk during the time specified by hour
   * Precondition: 0 <= hour <= 23
   */
  public int numAvailableDogs(int hour)
  { /* implementation not shown */ }

  /**
   * Decreases, by numberDogsWalked, the number of dogs available for a walk
   * during the time specified by hour
   * Preconditions: 0 <= hour <= 23
   *                numberDogsWalked > 0
   */
  public void updateDogs(int hour, int numberDogsWalked)
  { /* implementation not shown */ }

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

A dog walker is associated with a dog-walking company and is represented by the `DogWalker` class. You will write two methods of the `DogWalker` class.

```
public class DogWalker
{
  /** The maximum number of dogs this walker can walk simultaneously
      per hour */
  private int maxDogs;

  /** The dog-walking company this dog walker is associated with */
  private DogWalkCompany company;

  /**
   * Assigns max to maxDogs and comp to company
   * Precondition: max > 0
   */
  public DogWalker(int max, DogWalkCompany comp)
  { /* implementation not shown */ }

  /**
   * Takes at least one dog for a walk during the time specified by
   * hour, as described in part (a)
   * Preconditions: 0 <= hour <= 23
   *                maxDogs > 0
   */
  public int walkDogs(int hour)
  { /* to be implemented in part (a) */ }
```

```
  /**
   * Performs an entire dog-walking shift and returns the amount
   * earned, in dollars, as described in part (b)
   * Preconditions: 0 <= startHour <= endHour <= 23
   *                maxDogs > 0
   */
  public int dogWalkShift(int startHour, int endHour)
  { /* to be implemented in part (b) */ }

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

A. Write the `walkDogs` method, which updates and returns the number of dogs this dog walker walks during the time specified by `hour`. Values of `hour` range from `0` to `23`, inclusive.

A helper method, `numAvailableDogs`, has been provided in the `DogWalkCompany` class. The method returns the number of dogs available to be taken for a walk at a given hour. The dog walker will always walk as many dogs as the dog-walking company has available to walk, as long as the available number of dogs is not greater than the maximum number of dogs that the dog walker can handle, represented by the value of `maxDogs`.

Another helper method, `updateDogs`, has also been provided in the `DogWalkCompany` class. So that multiple dog walkers do not sign up to walk the same dogs, the `walkDogs` method should use `updateDogs` to update the dog-walking company with the number of dogs this dog walker will walk during the given hour. The parameters of the `updateDogs` method indicate how many dogs this dog walker will walk during the time specified by `hour`.

For example, if the dog-walking company has 10 dogs that need to be walked at the given hour but the dog walker's maximum is 4, the `updateDogs` method should be used to indicate that this dog walker will walk 4 of the 10 dogs during the given hour. As another example, if the dog-walking company has 3 dogs that need to be walked at the given hour and the dog walker's maximum is 4, the `updateDogs` method should be used to indicate that this dog walker will walk all 3 of the available dogs during the given hour.

The `walkDogs` method should return the number of dogs to be walked by this dog walker during the time specified by `hour`.

Complete method `walkDogs`. You must use `numAvailableDogs` and `updateDogs` appropriately to receive full credit.

```
/**
 * Takes at least one dog for a walk during the time specified by
 * hour, as described in part (a)
 * Preconditions: 0 <= hour <= 23
 *                maxDogs > 0
 */
public int walkDogs(int hour)
```

B. Write the `dogWalkShift` method, which performs a dog-walking shift of the hours in the range `startHour` to `endHour`, inclusive, and returns the total amount earned. For example, a dog-walking shift from `14` to `16` is composed of three one-hour dog walks starting at hours `14`, `15`, and `16`.

For each hour, the base pay is $5 per dog walked plus a bonus of $3 if at least one of the following is true.

- `maxDogs` dogs are walked
- the walk occurs between the peak hours of 9 and 17, inclusive

The following table shows an example of the calculated amount earned by walking dogs from hour 7 through hour 10, inclusive.

Hour	Maximum Number of Dogs	Number of Dogs Walked	Amount Earned, in Dollars
7	3	3	3 × 5 + 3 = 18
8	3	2	2 × 5 = 10
9	3	2	2 × 5 + 3 = 13
10	3	3	3 × 5 + 3 = 18
Total			59

Complete method `dogWalkShift`. Assume that `walkDogs` works as specified, regardless of what you wrote in part (a). You must use `walkDogs` appropriately to earn full credit.

```
/**
 * Performs an entire dog-walking shift and returns the amount
 * earned, in dollars, as described in part (b)
 * Preconditions: 0 <= startHour <= endHour <= 23
 *                maxDogs > 0
 */
public int dogWalkShift(int startHour, int endHour)
```

2. This question involves the `SignedText` class, which contains methods that are used to include a signature as part of a string of text. You will write the complete `SignedText` class, which contains a constructor and two methods.

The `SignedText` constructor takes two `String` parameters. The first parameter is a first name and the second parameter is a last name. The length of the second parameter is always greater than or equal to 1.

The `getSignature` method takes no parameters and returns a formatted signature string constructed from the first and last names according to the following rules.

- If the first name is an empty string, the returned signature string contains just the last name.
- If the first name is not an empty string, the returned signature string is the first letter of the first name, a dash `("-")`, and the last name, in that order.

The `addSignature` method returns a possibly revised copy of its `String` parameter. The parameter will contain at most one occurrence of the object's signature, at either the beginning or the end of the parameter. The returned string is created from the parameter according to the following rules.

- If the object's signature does not occur in the `String` parameter of the method, the returned `String` is the value of the parameter with the signature added to the end.
- If the object's signature occurs at the end of the `String` parameter, the returned `String` is the unchanged value of the parameter.
- If the object's signature occurs at the beginning of the `String` parameter, the returned `String` is the value of the original parameter with the signature removed from the beginning and appended to the end of the parameter.

The following table contains a sample code execution sequence and the corresponding results. The code execution sequence appears in a class other than `SignedText`.

Statement	Method Call Return Value (blank if none)	Explanation
`SignedText st1 = new SignedText("", "Wong");`		The `SignedText` object `st1` has an empty first name and last name `"Wong"`.
`String temp = st1.getSignature();`	`"Wong"`	
`SignedText st2 = new SignedText("henri", "dubois");`		The `SignedText` object `st2` has first name `"henri"` and last name `"dubois"`.
`temp = st2.getSignature();`	`"h-dubois"`	
`SignedText st3 = new SignedText("GRACE", "LOPEZ");`		The `SignedText` object `st3` has first name `"GRACE"` and last name `"LOPEZ"`.
`temp = st3.getSignature();`	`"G-LOPEZ"`	
`SignedText st4 = new SignedText("", "FOX");`		The `SignedText` object `st4` has an empty first name and last name `"FOX"`.
`String text = "Dear";`		
`temp = st4.addSignature(text);`	`"DearFOX"`	The signature does not occur in the `addSignature` parameter, so the returned string is the value of the parameter with the signature appended.
`text = "Best wishesFOX";`		
`temp = st4.addSignature(text);`	`"Best wishesFOX"`	The signature occurs at the end of the `addSignature` parameter, so the returned string is the unchanged value of the parameter.

Statement	Method Call Return Value (blank if none)	Explanation
`text = "FOXThanks";`		
`temp = st4.addSignature(text);`	`"ThanksFOX"`	The signature occurs at the beginning of the `addSignature` parameter, so the returned string is the value of the original parameter with the signature removed from the beginning and appended to the end of the parameter.
`text = "G-LOPEZHello";`		
`temp = st3.addSignature(text);`	`"HelloG-LOPEZ"`	The signature occurs at the beginning of the `addSignature` parameter, so the returned string is the value of the original parameter with the signature removed from the beginning and appended to the end of the parameter.

Write the complete `SignedText` class. Your implementation must meet all specifications and conform to the examples in the table.

3. This question involves pairing competitors in a tournament into one-on-one matches for one round of the tournament. For example, in a chess tournament, the competitors are the individual chess players. A game of chess involving two players is a match. The winner of each match goes on to a match in the next round of the tournament. Since half of the players are eliminated in each round of the tournament, there is eventually a final round consisting of one match and two competitors. The winner of that match is considered the winner of the tournament.

Competitors, matches, and rounds of the tournament are represented by the `Competitor`, `Match`, and `Round` classes. You will write the constructor and one method of the `Round` class.

```
/** A single competitor in the tournament */
public class Competitor
{
  /** The competitor's name and rank */
  private String name;
  private int rank;

  /**
   * Assigns n to name and initialRank to rank
   * Precondition: initialRank >= 1
   */
  public Competitor(String n, int initialRank)
  { /* implementation not shown */ }

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

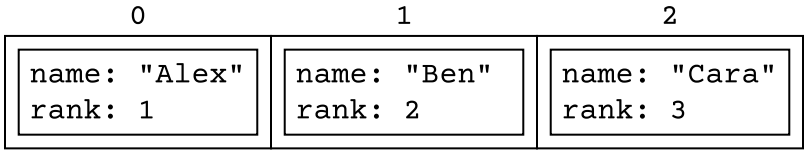
```
/** A match between two competitors */
public class Match
{
  public Match(Competitor one, Competitor two)
  { /* implementation not shown */ }

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


/** A single round of the tournament */
public class Round
{
  /** The list of competitors participating in this round */
  private ArrayList<Competitor> competitorList;

  /** Initializes competitorList, as described in part (a) */
  public Round(String[] names)
  { /* to be implemented in part (a) */ }

  /**
   * Creates an ArrayList of Match objects for the next round
   * of the tournament, as described in part (b)
   * Preconditions: competitorList contains at least one element.
   *                competitorList is ordered from best to worst rank.
   * Postcondition: competitorList is unchanged.
   */
  public ArrayList<Match> buildMatches()
  { /* to be implemented in part (b) */ }

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

A. Write the constructor for the `Round` class. The constructor should initialize `competitorList` to contain one `Competitor` object for each name in the `String[] names`. The order in which `Competitor` objects appear in `competitorList` should be the same as the order in which they appear in `names`, and the rank of each competitor is based on the competitor's position in `names`. Names are listed in `names` in order from the best-ranked competitor with rank `1` to the worst-ranked competitor with rank *n*, where *n* is the number of elements in `names`.

For example, assume the following code segment is executed.

```
String[] players = {"Alex", "Ben", "Cara"};
Round r = new Round(players);
```

The following shows the contents of `competitorList` in `r` after the constructor has finished executing.



Complete the `Round` constructor.

```
/** Initializes competitorList, as described in part (a) */
public Round(String[] names)
```

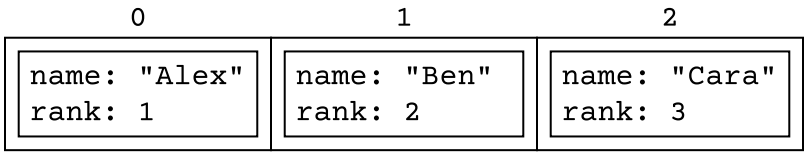
B. Write the `Round` method `buildMatches`. This method should return a new `ArrayList<Match>` object by pairing competitors from `competitorList` according to the following rules.

- If the number of competitors in `competitorList` is even, the best-ranked competitor is paired with the worst-ranked competitor, the second-best-ranked competitor is paired with the second-worst-ranked competitor, etc.
- If the number of competitors in `competitorList` is odd, the competitor with the best rank is ignored and the remaining competitors are paired according to the rule for an even number of competitors.

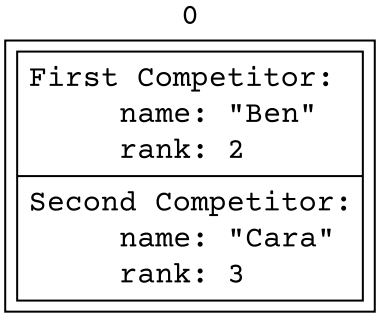
Each pair of competitors is used to create a `Match` object that should be added to the `ArrayList` to return. Competitors may appear in either order in a `Match` object, and matches may appear in any order in the returned `ArrayList`.

The following example shows the contents of `competitorList` in a `Round` object `r1` containing an odd number of competitors and the `ArrayList` of `Match` objects that should be returned by the call `r1.buildMatches()`.

`competitorList`:

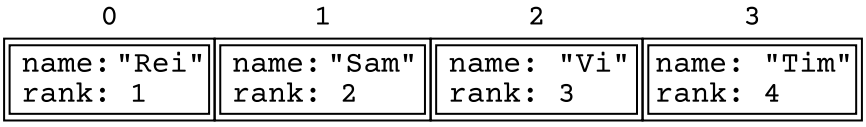


The `ArrayList` to be returned contains a single match between Ben and Cara, the second and third-ranked competitors:

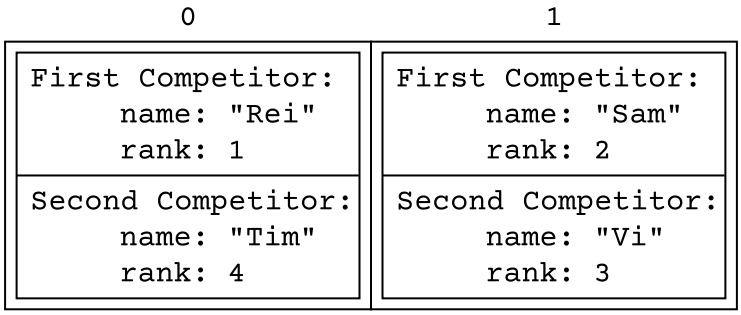


The next example shows the contents of `competitorList` in a `Round` object `r2` containing an even number of competitors and the `ArrayList` of `Match` objects that should be returned by the call `r2.buildMatches()`.

`competitorList`:



The `ArrayList` to be returned contains two matches: one match between the first- and last-ranked competitors Rei and Tim, and a second match between the second- and third-ranked competitors Sam and Vi:



Complete the `buildMatches` method.

```
/**
 * Creates an ArrayList of Match objects for the next round
 * of the tournament, as described in part (b)
 * Preconditions: competitorList contains at least one element.
 *                competitorList is ordered from best to worst rank.
 * Postcondition: competitorList is unchanged.
 */
public ArrayList<Match> buildMatches()
```

4. This question involves reasoning about a number puzzle that is represented as a two-dimensional array of integers. Each element of the array initially contains a value between 1 and 9, inclusive. Solving the puzzle involves clearing pairs of array elements by setting them to 0. Two elements can be cleared if their values sum to 10 or if they have the same value. The puzzle is considered solved if all elements of the array are cleared.

You will write the constructor and one method of the `SumOrSameGame` class, which contains the methods that manipulate elements of the puzzle.

```
public class SumOrSameGame
{
  private int[][] puzzle;

  /**
   * Creates a two-dimensional array and fills it with random integers,
   * as described in part (a)
   * Precondition: numRows > 0; numCols > 0
   */
  public SumOrSameGame(int numRows, int numCols)
  { /* to be implemented in part (a) */ }

  /**
   * Identifies and clears an element of puzzle that can be paired with
   * the element at the given row and column, as described in part (b)
   * Preconditions: row and col are valid row and column indices in puzzle.
   *  The element at the given row and column is between 1 and 9, inclusive.
   */
  public boolean clearPair(int row, int col)
  { /* to be implemented in part (b) */ }

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

A. Write the constructor for the `SumOrSameGame` class. The constructor initializes the instance variable `puzzle` to be a two-dimensional integer array with the number of rows and columns specified by the parameters `numRows` and `numCols`, respectively. Array elements are initialized with random integers between `1` and `9`, inclusive, each with an equal chance of being assigned to each element of `puzzle`.

When an element of the two-dimensional array is accessed, the first index is used to specify the row and the second index is used to specify the column.

Complete the `SumOrSameGame` constructor.

```
/**
 * Creates a two-dimensional array and fills it with random integers,
 * as described in part (a)
 * Precondition: numRows > 0; numCols > 0
 */
public SumOrSameGame(int numRows, int numCols)
```

B. Write the `clearPair` method, which takes a valid row index and valid column index as its parameters. The array element specified by those indices, which has a value between `1` and `9`, inclusive, is compared to other array elements in `puzzle` in an attempt to pair it with another array element that meets both of the following conditions.

- The row index of the second element is greater than or equal to the parameter `row`.
- The two elements have equal values or have values that sum to 10.

If such an array element is found, both array elements of the pair are cleared (set to `0`) and the method returns `true`. If more than one such array element is found, any one of those identified array elements can be used to complete the pair and can be cleared. If no such array element is found, no changes are made to `puzzle` and the method returns `false`.

The following table shows the possible results of several calls to `clearPair`.

`puzzle` before call to `clearPair`	Method Call	`puzzle` after call to `clearPair`	Return Value	Explanation

`0 7 9 0`
`7 4 1 6`
`8 4 1 8`
	`clearPair(0, 1)`	`0 0 9 0`
		`0 4 1 6`
		`8 4 1 8`
			`true`	The value `7` in row `0`, column `1` is matched with the value `7` in row `1`, column `0`.

`1 2 3 4`
`5 6 7 8`
`5 4 1 2`
	`clearPair(2, 2)`	`1 2 3 4`
		`5 6 7 8`
		`5 4 1 2`
			`false`	There is no element in row `2` or a later row to pair with the value `1`.

`8 1 0 5`
`0 4 3 6`
`3 4 5 8`
	`clearPair(1, 1)`	`8 1 0 5`
		`0 0 3 0`
		`3 4 5 8`
			`true`	The value `4` in row `1`, column `1` is matched with the value `6` in row `1`, column `3`. It could also have been matched with the value `4` in row `2`, column `1`.

`1 7 9`
`2 6 5`
`4 4 4`
	`clearPair(0, 2)`	`0 7 0`
		`2 6 5`
		`4 4 4`
			`true`	The value `9` in row `0`, column `2` is matched with the value `1` in row `0`, column `0`.

Complete method `clearPair`.

```
/**
 * Identifies and clears an element of puzzle that can be paired with
 * the element at the given row and column, as described in part (b)
 * Preconditions: row and col are valid row and column indices in puzzle.
 *  The element at the given row and column is between 1 and 9, inclusive.
 */
public boolean clearPair(int row, int col)
```

STOP

END OF EXAM