CollegeBoard
Advanced Placement
Program

AP® Computer Science AB
2007 Free-Response Questions

COMPUTER SCIENCE AB
SECTION II

Time—1 hour and 45 minutes

Number of questions—4

Percent of total grade—50

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

Notes:

- Assume that the classes listed in the Quick Reference found in the Appendix have been imported where appropriate.
- The `java.util.Stack` and `java.util.PriorityQueue` classes and the `java.util.Queue` interface (page A2 in the Appendix) each inherit methods that access elements in a way that violates their abstract data structure definitions. Solutions that use objects of types `Stack`, `Queue`, and `PriorityQueue` should use only the methods listed in the Appendix for accessing and modifying those objects. The use of other methods may not receive full credit.
- Assume that the implementation classes `ListNode` and `TreeNode` (page A4 in the Appendix) are used for any questions referring to linked lists or trees, unless otherwise specified.
- `ListNode` and `TreeNode` parameters may be `null`. Otherwise, unless 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.
- When Big-Oh running time is required for a response, you must use the most restrictive Big-Oh expression. For example, if the running time is $O(n)$, a response of $O(n^2)$ will not be given credit.

GO ON TO THE NEXT PAGE.

1. The sliding puzzle is a popular puzzle toy and can be represented as a square grid in which all but one location contains a numbered tile. For example, the following diagram shows a 4×4 puzzle containing tiles numbered 1 through 15 and a single "hole."

1	4	3	12
11		5	7
13	14	6	2
10	8	15	9

The puzzle is solved by sliding tiles until all numbers are arranged in numerical sequence when traversed in row-major order, as shown in the following diagram. Note that the hole may be in any position.

1	2	3	4
5	6	7	
8	9	10	11
12	13	14	15

The puzzle is represented by the `SlidingPuzzle` class shown below. In this question, you will implement two methods of this class.

```
public class SlidingPuzzle
{
  private int side;         // the side length of the puzzle grid
  private int[][] values;   // the tile values with the hole represented by 0

  /** @param sideLength the side length of the square grid
   *            Precondition: sideLength > 0
   */
  public SlidingPuzzle(int sideLength)
  {
    side = sideLength;
    values = new int[side][side];
    initialize();
  }
```

GO ON TO THE NEXT PAGE.

```
    /** Precondition: the puzzle grid contains the distinct values 0 through side^2 – 1
     *  @return true  if the tiles in the puzzle are all arranged in increasing order
     *                (the hole value 0 may be in any position);
     *          false  otherwise
     */
    public boolean isDone()
    {  /* to be implemented in part (a) */  }


    /** Initializes the puzzle by placing numbers 0 through side^2 – 1 into random locations
     */
    public void initialize()
    {  /* to be implemented in part (b) */  }

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

(a) Write the `SlidingPuzzle` method `isDone`. Method `isDone` returns `true` if the values in the puzzle appear in increasing order when traversed in row-major order; otherwise, it returns `false`. The value 0 (denoting the hole) may appear anywhere within the puzzle.

Complete method `isDone` below.

```
    /** Precondition: the puzzle grid contains the distinct values 0 through side^2 – 1
     *  @return true  if the tiles in the puzzle are all arranged in increasing order
     *                (the hole value 0 may be in any position);
     *          false  otherwise
     */
    public boolean isDone()
```

GO ON TO THE NEXT PAGE.

(b) Write the `SlidingPuzzle` method `initialize`. Method `initialize` fills the `side` × `side` `values` grid with random integers 0 through $\texttt{side}^2 - 1$, without repeating numbers.

Your implementation must use the following algorithm.

1. Initialize an `ArrayList<Integer>` named `temp` with values 0 through $n = \texttt{side}^2 - 1$. The code for this step is provided.

Starting with the first element of the grid `values`, repeat steps 2 and 3 until the grid has been filled.

2. Pick a random element from `temp` and place that element into the next empty grid location.
3. Remove that element from `temp` by calling the `ArrayList remove` method on its index.

Complete method `initialize` below. The method has been started for your convenience.

```
/** Initializes the puzzle by placing numbers 0 through side^2 – 1 into random locations
 */
public void initialize()
{
  ArrayList<Integer> temp = new ArrayList<Integer>();
  for (int j = 0; j < side * side; j++)
    temp.add(new Integer(j));

  //   Write your solution below.











}
```

GO ON TO THE NEXT PAGE.

(c) What is the expected big-Oh running time of the initialization algorithm described in part (b), <u>in terms of n, the number of tiles</u>?

(d) Consider a variation of the algorithm described in part (b) in which Step 3 is changed.

1. Initialize an `ArrayList<Integer>` named `temp` with values 0 through $n = \texttt{side}^2 - 1$.

Starting with the first element of the grid `values`, repeat steps 2 and 3 until the grid has been filled.

2. Pick a random element from `temp` and place that element into the next empty grid location.
3. Replace that element with the element in the last index of `temp`, then remove the last element from `temp`. (Note: removing the last item of an `ArrayList` is a constant time operation.)

What is the expected big-Oh running time of this variation <u>in terms of n, the number of tiles</u>?

GO ON TO THE NEXT PAGE.

2. Consider a system for choosing pairs of people to be lab partners based on a compatibility score. A `Person` class (whose implementation is not shown) is used by the `Pair` class and the `PairMatcher` class. The `Person` class implements appropriate `hashCode` and `equals` methods.

   The declaration for the `Pair` class is shown below. A `Pair` object is constructed with two `Person` objects and has a method that calculates and compares the compatibility scores of two `Pair` objects.

```
public class Pair implements Comparable
{
  /** @param p1 the first Person of the Pair
   *  @param p2 the second Person of the Pair
   */
  public Pair(Person p1, Person p2)
  {  /* implementation not shown */  }

  /** @return first Person of this Pair
   */
  public Person getPerson1()
  {  /* implementation not shown */  }

  /** @return second Person of this Pair
   */
  public Person getPerson2()
  {  /* implementation not shown */  }

  /** @param other the object to be compared to this Pair
   *           Precondition: other is a Pair object
   *  @return the result of the comparison of the compatibility scores of this Pair and other
   */
  public int compareTo(Object other)
  {  /* implementation not shown */  }

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

GO ON TO THE NEXT PAGE.

The `PairMatcher` class declared below will be used to maintain compatibility information about a group of people. A `PairMatcher` object is constructed with a list of `Person` objects. The constructor creates a mapping between each `Person` in the list and a priority queue of pairings between that `Person` and every other `Person` in the list.

```
public class PairMatcher
{
  private Map<Person, PriorityQueue<Pair>> personMap;


  /** Initializes and fills personMap so that each Person in personList is a key,
   *  and the value associated with each key k is a PriorityQueue of Pair objects
   *  pairing k with all other Persons in personList
   *  @param personList a nonempty list of Person objects
   */
  public PairMatcher(List<Person> personList)
  {  /* to be implemented in part (a) */  }


  /** @param p the Person to be matched
   *  @param num the number of Person objects to remove
   *          Precondition: if p is in personMap, then num is > 0 and less than or equal to
   *                        the number of pairs in the priority queue associated with p
   *  @return an array of the num removed Person objects;
   *           null if p is not in personMap
   */
  public Person[] removeNumMatches(Person p, int num)
  {  /* to be implemented in part (b) */  }


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

GO ON TO THE NEXT PAGE.

(a) Write the `PairMatcher` constructor. The constructor builds `personMap` such that each `Person` object in `personList` is a key in the `Map`. The value associated with each key *k* is a `PriorityQueue` of `Pair` objects such that for all *p* in `personList`, if *p* is not equal to *k*, there is a `Pair` in which *k* is the first `Person` and *p* is the second `Person`.

For example, the following shows a list of `Person` objects and the map that would be created as a result of constructing a `PairMatcher` object with that list. The `Pair` objects in the map show the first `Person`, the second `Person`, and the compatibility score (lower scores mean better compatibility).

`personList`

Jamie	Chris	Pat	Terry

`personMap`

Jamie	:	Jamie Pat (10)	Jamie Chris (11)	Jamie Terry (22)
Chris	:	Chris Jamie (11)	Chris Terry (11)	Chris Pat (21)
Pat	:	Pat Jamie (10)	Pat Chris (21)	Pat Terry (32)
Terry	:	Terry Chris (11)	Terry Jamie (22)	Terry Pat (32)

Complete the `PairMatcher` constructor below.

```
/** Initializes and fills personMap so that each Person in personList is a key,
 *  and the value associated with each key k is a PriorityQueue of Pair objects
 *  pairing k with all other Persons in personList
 *  @param personList a nonempty list of Person objects
 */
public PairMatcher(List<Person> personList)
```

GO ON TO THE NEXT PAGE.

(b) Write the `PairMatcher` method `removeNumMatches`. Method `removeNumMatches` removes the first `num` `Pair` objects from the priority queue associated with `p` in `personMap` and returns an array containing the second `Person` of each `Pair` that was removed. The `Person` objects in the returned array should be ordered by their compatibility with `Person p`. If `Person p` is not in the map, `null` is returned.

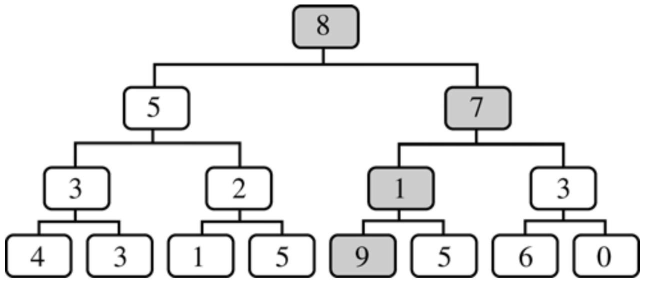
Complete method `removeNumMatches` below.

```
/** @param p  the Person to be matched
 *  @param num  the number of Person objects to remove
 *              Precondition: if p is in personMap, then num is > 0 and less than or equal to
 *                            the number of pairs in the priority queue associated with p
 *  @return  an array of the num removed Person objects;
 *           null if p is not in personMap
 */
public Person[] removeNumMatches(Person p, int num)
```

GO ON TO THE NEXT PAGE.

3. A game called TreeBall has been created in which a player drops a ball into the top of the game board. The score is computed by summing the values in the nodes along the path that the ball follows.

   The game board for TreeBall is a full binary tree in which the root represents the top of the game board. For this question, assume that a full binary tree is one in which all leaves are at the same level and all non-leaf nodes have exactly two children. Each node in the tree contains an integer from 0 to 9. The diagram below shows an example TreeBall game board with 4 levels. In this example, the path with the greatest total is shown by the shaded nodes from the root (8) to the leaf (9). The score for this path is 25 = 8 + 7 + 1 + 9.



   The declaration for the `GameBoard` class is shown below.

```
public class GameBoard
{
  private TreeNode root;    //  the root of the tree


  /**  Creates a full binary tree rooted at  root  with  numLevels  levels
   *    with a random integer from 0 to 9, inclusive, generated for each node
   *   @param numLevels  the number of levels in the tree
   *              Precondition:  numLevels > 0
   */
  public GameBoard(int numLevels)
  {   /*  to be implemented in part (b)  */   }


  /** @return  the maximum path score for this  GameBoard
   */
  public int getMaxScore()
  {  return getMaxHelper(root);   }


  /** @param current  the root of the subtree to be processed
   *   @return  the maximum path score for the subtree rooted at  current
   */
  private int getMaxHelper(TreeNode current)
  {   /*  to be implemented in part (a)  */   }


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

(a) Write the `GameBoard` method `getMaxHelper`, which returns the maximum path score that can be obtained from the tree rooted at `current`. A path score is computed by summing the node values along the path from `current` to a leaf node. The maximum path score for an empty tree is 0.

Complete method `getMaxHelper` below.

```
/** @param current  the root of the subtree to be processed
 *  @return  the maximum path score for the subtree rooted at current
 */
private int getMaxHelper(TreeNode current)
```

(b) Write the `GameBoard` constructor, which creates a full binary tree with `numLevels` levels. Each node in the tree should contain an independently generated random `Integer` value from 0 to 9, inclusive. Recall that for this question a full binary tree has the property that all leaves are on the same level. You may find it useful to write and use a helper method to create the tree.

Complete the `GameBoard` constructor below.

```
/** Creates a full binary tree rooted at root with numLevels levels
 *  with a random integer from 0 to 9, inclusive, generated for each node
 *  @param numLevels  the number of levels in the tree
 *           Precondition: numLevels > 0
 */
public GameBoard(int numLevels)
```

GO ON TO THE NEXT PAGE.

4. This question involves reasoning about the code from the Marine Biology Simulation case study. A copy of the code is provided as part of this exam.

   Suppose that you want to visit all locations of a square environment in a single loop. There is a familiar interface for this purpose: the *iterator*. In this question, you will complete a method in a class `EnvIterator` that implements the `Iterator` interface.

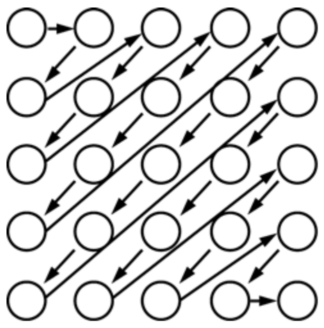
   Here is an incomplete definition of the `EnvIterator` class.

```
public class EnvIterator implements Iterator<Location>
{
  private Environment env; // the environment over which to iterate
  private Location loc;    // the next location to be returned


  /** @param anEnv the environment over which to iterate
   *            Precondition: anEnv is square, i.e., anEnv.numRows() == anEnv.numCols()
   */
  public EnvIterator(BoundedEnv anEnv)
  {
    env = anEnv;
    loc = new Location(0, 0);
  }


  /** @return true if this EnvIterator has more elements
   *           false otherwise
   */
  public boolean hasNext()
  {  return env.isValid(loc);  }


  /** Precondition:  hasNext() returns true
   *   Postcondition:  loc has been updated to the successor location
   *   @return the next location in the environment
   */
  public Location next()
  {   /* to be implemented in part (a) */   }


  /** Throws an UnsupportedOperationException since it is impossible to
   *   remove a location from an environment.
   */
  public void remove()
  {  throw new UnsupportedOperationException();  }
}
```

GO ON TO THE NEXT PAGE.

(a) Write the `EnvIterator` method `next`. The `next` method should return the next *location* in the environment, that is, the location that is referenced in the `loc` instance field when the method is called. The `next` method should also update the `loc` instance field to the successor location, as described below. Note that the first location returned by `next` is (0, 0) as initialized in the `EnvIterator` constructor.

Your implementation of `next` should allow the iterator to visit all elements of a square `BoundedEnv`, following the diagonal pattern shown in the diagram below.



The following describes the algorithm for determining the successor location in the diagonal pattern shown in the diagram above.

1. If the current location is at the *bottom* edge of the environment, move to the top of the next diagonal. For example, in the diagram given above, (4, 1) is followed by (2, 4).
2. Otherwise, if the current location is at the *left* edge of the environment, move to the top of the next diagonal. For example, in the diagram given above, (1, 0) is followed by (0, 2).
3. Otherwise, move down and left. For example, in the diagram given above, (1, 1) is followed by (2, 0).

Complete method `next` below.

```
/** Precondition: hasNext() returns true
 *  Postcondition: loc has been updated to the successor location
 *  @return the next location in the environment
 */
public Location next()
```

GO ON TO THE NEXT PAGE.

(b) A client class contains the method `emptyLocs`, which returns a list of the first `n` empty locations when a given square environment `env` is traversed by an `EnvIterator`. If there are fewer than `n` empty locations in `env`, `emptyLocs` should return all of them.

For example, suppose the environment `env` is as shown in the diagram below where `x` indicates an occupied location. In this example, the call `emptyLocs(env, 5)` returns a list of locations [(0, 1), (1, 0), (2, 0), (0, 3), (1, 2)].

	0	1	2	3
0	x		x	
1		x		x
2		x		
3		x		

Complete method `emptyLocs` below.

```
/** @param env  the environment over which to iterate
 *           Precondition: env is square, i.e., env.numRows() == env.numCols()
 *  @param n  the desired number of empty locations to be returned
 *           Precondition: n > 0
 *  @return  a list of the first n empty locations;
 *             all empty locations if there are fewer than n empty locations.
 *             Locations are ordered in the order in which they are visited by the EnvIterator
 */
public List<Location> emptyLocs(BoundedEnv env, int n)
```

STOP

END OF EXAM