AP® COMPUTER SCIENCE A
2008 SCORING GUIDELINES

Question 1: Flight List

Part A:	getDuration	**4 points**

+1 handle empty case
- **+1/2** check if `flights` is empty
- **+1/2** return 0 if empty

+1 access start time
- **+1/2** access `flights.get(0)`
- **+1/2** correctly call `getDepartureTime` on a flight

+1 access end time
- **+1/2** access `flights.get(flights.size()-1)`
- **+1/2** correctly call `getArrivalTime` on a flight

+1 calculate and return duration
- **+1/2** call `minutesUntil` using `Time` objects
- **+1/2** return correct duration (using `minutesUntil`)

Part B:	getShortestLayover	**5 points**

+1 handle case with 0 or 1 flight
- **+1/2** check if `flights.size() < 2`
- **+1/2** return -1 in that case

+1 traverse `flights`
- **+1/2** correctly access an element of `flights` (in context of loop)
- **+1/2** access all elements of `flights` (lose this if index out-of-bounds)

+2 1/2 find shortest layover (in context of loop)
- **+1** get layover time between successive flights (using `minutesUntil`)
- **+1/2** compare layover time with some previous layover
- **+1** correctly identify shortest layover

+1/2 return shortest layover

AP® COMPUTER SCIENCE A
2008 CANONICAL SOLUTIONS

Question 1: Flight List

PART A:

```
public int getDuration()
{
  if (flights.size() == 0)
  {
    return 0;
  }
  else
  {
    Time start = flights.get(0).getDepartureTime();
    Time end = flights.get(flights.size()-1).getArrivalTime();
    return start.minutesUntil(end);
  }
}
```

PART B:

```
public int getShortestLayover()
{
  if (flights.size() < 2)
  {
    return -1;
  }
  else
  {
    int shortest = getDuration();
    for (int i = 0; i < flights.size()-1; i++)
    {
      Time arrive = flights.get(i).getArrivalTime();
      Time leave = flights.get(i+1).getDepartureTime();
      int layover = arrive.minutesUntil(leave);
      if (layover < shortest)
      {
        shortest = layover;
      }
    }
    return shortest;
  }
}
```

A1A,

```
public class Trip
{
  private ArrayList<Flight> flights;
     // stores the flights (if any) in chronological order

  /** @return the number of minutes from the departure of the first flight to the arrival
   *          of the last flight if there are one or more flights in the trip;
   *          0, if there are no flights in the trip
   */
  public int getDuration()
  {   /* to be implemented in part (a) */   }

  /** Precondition: the departure time for each flight is later than the arrival time of its
   *               preceding flight
   *   @return the smallest number of minutes between the arrival of a flight and the departure
   *           of the flight immediately after it, if there are two or more flights in the trip;
   *           -1, if there are fewer than two flights in the trip
   */
  public int getShortestLayover()
  {   /* to be implemented in part (b) */   }

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

(a) Complete method `getDuration` below.

```
  /** @return the number of minutes from the departure of the first flight to the arrival
   *          of the last flight if there are one or more flights in the trip;
   *          0, if there are no flights in the trip
   */
  public int getDuration()
{
return((flights.get(0)).getDepartureTime()).minutesUntil(((flights.get(
     flights.size()-1)).getArrivalTime());
}
```

Part (b) begins on page 6.

GO ON TO THE NEXT PAGE.

Complete method `getShortestLayover` below.

```
/** Precondition: the departure time for each flight is later than the arrival time of its
 *                 preceding flight
 *  @return  the smallest number of minutes between the arrival of a flight and the departure
 *           of the flight immediately after it, if there are two or more flights in the trip;
 *           -1, if there are fewer than two flights in the trip
 */
public int getShortestLayover()
{ if(flights.size() < 2)
    {return -1;}
  int small = flights.get(0).getArrivalTime().minutesUntil(
              flights.get(1).getDepartureTime());

  for(int index=1; index<flights.size()-1; index++)
  { if(flights.get(index).getArrivalTime().minutesUntil(
          flights.get(index+1).getDepartureTime() < small)
      { small = flights.get(index).getArrivalTime().minutesUntil(
              flights.get(index+1).getDepartureTime();
      }
  }
  return small;
}
```

GO ON TO THE NEXT PAGE.

A1 B,

```
public class Trip
{
   private ArrayList<Flight> flights;
      // stores the flights (if any) in chronological order

   /** @return the number of minutes from the departure of the first flight to the arrival
    *          of the last flight if there are one or more flights in the trip;
    *          0, if there are no flights in the trip
    */
   public int getDuration()
   {  /* to be implemented in part (a) */  }

   /** Precondition: the departure time for each flight is later than the arrival time of its
    *                preceding flight
    *  @return the smallest number of minutes between the arrival of a flight and the departure
    *          of the flight immediately after it, if there are two or more flights in the trip;
    *          -1, if there are fewer than two flights in the trip
    */
   public int getShortestLayover()
   {  /* to be implemented in part (b) */  }

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

(a) Complete method getDuration below.

```
/** @return ~~the number of minutes~~ from the departure of the first flight to the arrival
 *          of the last flight ~~if there are one or more flights in the trip~~;
 *          ~~0, if there are no flights in the trip~~
 */
public int getDuration(){
int minutes = 0;
Time x = flights.get(0).getDepartureTime();
int last = flights.size()-1;
Time y=flights.get(last).getArrivalTime();
if (flights.size() >= 1)
     minutes += x.minutesUntil(y);
        return minutes;

else
return 0;
}
```

Part (b) begins on page 6.

GO ON TO THE NEXT PAGE.

A1B2

Complete method `getShortestLayover` below.

```
/**. Precondition: the departure time for each flight is later than the arrival time of its
 *                 preceding flight
 *   @return  the smallest number of minutes between the arrival of a flight and the departure
 *            of the flight immediately after it, if there are two or more flights in the trip;
 *            -1, if there are fewer than two flights in the trip
 */
public int getShortestLayover() {

private ArrayList<Time> arrayOfTimes = new ArrayList<Time>()

int layoverTime = 0;
for(int i=0; i<flights.size(); i++) {
    int temp = 0;
    temp=flights.get(i).getArrivalTime().minutesUntil(
        flights.get(i+1).getDepartureTime());
    arrayOfTimes.add(temp);
}
if (flights.size() >= 2)

        for (int k = 1; k < arrayOfTimes.size(); k++) {

            if (arrayOfTimes.get(k) > arrayOfTimes.get(k-1))

                layoverTime = arrayOfTimes.get(k);

                    return layoverTime;

        }
else
return -1;

}
```

GO ON TO THE NEXT PAGE.

A1C,

```
public class Trip
{
  private ArrayList<Flight> flights;
     // stores the flights (if any) in chronological order

  /** @return the number of minutes from the departure of the first flight to the arrival
   *         of the last flight if there are one or more flights in the trip;
   *         0, if there are no flights in the trip
   */
  public int getDuration()
  {  /* to be implemented in part (a) */  }

  /** Precondition: the departure time for each flight is later than the arrival time of its
   *              preceding flight
   *  @return the smallest number of minutes between the arrival of a flight and the departure
   *         of the flight immediately after it, if there are two or more flights in the trip;
   *         -1, if there are fewer than two flights in the trip
   */
  public int getShortestLayover()
  {  /* to be implemented in part (b) */  }

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

(a) Complete method `getDuration` below.

```
  /** @return the number of minutes from the departure of the first flight to the arrival
   *         of the last flight if there are one or more flights in the trip;
   *         0, if there are no flights in the trip
   */
  public int getDuration()
{    int Duration=0;
  if ( flights. length > 0)
  {
  Duration= flights[flights.length +1]. getArrivalTime() - flights[0]. get
  DepartureTime();
   return Duration;
  }
else if ( flights. length == 0 )
   return 0;
```

Part (b) begins on page 6.

GO ON TO THE NEXT PAGE.

A1C2

Complete method `getShortestLayover` below.

```
/** Precondition: the departure time for each flight is later than the arrival time of its
 *                  preceding flight
 *  @return  the smallest number of minutes between the arrival of a flight and the departure
 *           of the flight immediately after it, if there are two or more flights in the trip;
 *           -1, if there are fewer than two flights in the trip
 */
public int getShortestLayover()
{     int temp=0;
      int n=0;

    for (n ; n < flights.length; n++)
    {
          for (int j=0; j < flights.length; j++)
          {
             if((flight[j].get
```

GO ON TO THE NEXT PAGE.

Question 1

Overview

This question focused on abstraction, `ArrayList` traversal, and the application of basic algorithms. Students were provided with the frameworks of two helper classes: a `Time` class for representing a specific time, and a `Flight` class for representing an airline flight between cities. They were then asked to implement two methods of a third `Trip` class, which stores a sequence of `Flight` objects in an `ArrayList` instance variable. In part (a) students were required to implement the `getDuration` method for determining the length of a trip. This could be accomplished by calling the appropriate `Flight` and `Time` methods on the first and last flights in the `ArrayList` instance variable. In part (b) students were required to implement the `shortestLayover` method for finding the shortest layover between flights on the trip. This involved traversing the `ArrayList` of flights, determining the layover between successive flights (by calling the appropriate `Flight` and `Time` methods), and identifying the minimum layover duration.

Sample: A1A
Score: 8

In part (a) the student does not check the data structure for a size of zero and does not return a 0 if the structure is empty. The student then correctly accesses the departure time of the first flight by calling `getDepartureTime` correctly on the first element of the `ArrayList` of flights and correctly accesses the arrival time of the final flight by calling `getArrivalTime` on the final element of the `ArrayList` of flights. The student correctly uses the returned `Time` objects as parameters in the call to the `minutesUntil` method that will return the correct answer for this solution.

In part (b) the student correctly checks the data structure for a size of at least two flights, which is needed to create a layover between flights, and the student correctly returns a -1 if the structure was too small. The student then correctly accesses all of the elements of the data structure. The student uses these elements to correctly get a layover time between two flights, compares this layover time with the smallest time so far, and then correctly saves the layover time if it is smaller. The student then returns the smallest calculated layover.

Sample: A1B
Score: 6

In part (a) the student checks the data structure for a size of zero too late but does return a 0 if the structure is empty. The student then correctly accesses the departure time of the first flight by calling `getDepartureTime` on the first element of the `ArrayList` of flights and correctly accesses the arrival time of the final flight by calling `getArrivalTime` on the final element of the `ArrayList` of flights. The student stores each returned `Time` object and correctly uses those objects to call the `minutesUntil` method that will return the correct answer for this solution.

In part (b) the student checks the data structure for a size of at least two flights too late but does return -1 when the structure is too small. The student does not correctly access all of the elements of the data structure because there is an off by one error in the first loop. The student uses these elements correctly to get a layover time between two flights and compares this layover time with the smallest time so far. Because the return is inside the loop, however, the student does not correctly calculate or return the smallest time.

Question 1 (continued)

Sample: A1C
Score: 2

In part (a) the student checks the data structure for a size of zero and correctly returns a 0 based on that check. The student then incorrectly accesses the departure time of the first flight by calling `getDepartureTime` correctly but on an incorrectly accessed first element of the `ArrayList` of flights. The student also incorrectly accesses the arrival time of the final flight by calling `getArrivalTime` on an incorrect attempt to access the final element of the `ArrayList` of flights. The student incorrectly calculates the duration of the trip by trying to subtract the times instead of calling the `minutesUntil` method in the `Time` class.

The student does not write code that can solve part (b).