AP® COMPUTER SCIENCE A
2012 GENERAL SCORING GUIDELINES

Apply the question-specific rubric first, which always takes precedence. Penalty points can only be deducted in a part of the question that has earned credit via the question-specific rubric. No part of a question — (a), (b), or (c) — may have a negative point total. A given penalty can be assessed only once for a question, even if it occurs multiple times or in different parts of that question.

1-Point Penalty

(w) Extraneous code that causes a side effect or prevents earning points in the rubric (*e.g., information written to output*)

(x) Local variables used but none declared

(y) Destruction of persistent data *(e.g., changing value referenced by parameter)*

(z) `Void` method or constructor that returns a value

No Penalty

- Extraneous code that causes no side effect
- Extraneous code that is unreachable and would not have earned points in rubric
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared, provided that other variables are declared in some part
- `private` qualifier on local variable
- Missing `public` qualifier on class or constructor header
- Keyword used as an identifier
- Common mathematical symbols used for operators (x • ÷ ≤ ≥ < > ≠)
- `[]` vs. `()` vs. `<>`
- `=` instead of `==` (and vice versa)
- Array/collection element access confusion (`[]` vs. `get` for r-values)
- Array/collection element modification confusion (`[]` vs. `set` for l-values)
- `length/size` confusion for array, `String`, and `ArrayList`, with or without `()`
- Extraneous `[]` when referencing entire array
- `[i,j]` instead of `[i][j]`
- Extraneous size in array declaration, (*e.g.,* `int[size] nums = new int[size];`)
- Missing `;` provided that line breaks and indentation clearly convey intent
- Missing `{ }` where indentation clearly conveys intent and `{ }` are used elsewhere
- Missing `( )` on parameter-less method or constructor invocations
- Missing `( )` around `if/while` conditions
- Use of local variable outside declared scope (must be within same method body)
- Failure to cast object retrieved from nongeneric collection

* *Spelling and case discrepancies for identifiers fall under the "No Penalty" category only if the correction can be* **unambiguously** *inferred from context; for example, "*`ArayList`*" instead of "*`ArrayList`*". As a counterexample, note that if the code declares "*`Bug bug;`*" and then uses "*`Bug.move()`*" instead of "*`bug.move()`*", the context does* **not** *allow for the reader to assume the object instead of the class.*

AP® COMPUTER SCIENCE A
2012 SCORING GUIDELINES

Question 1: Climbing Club

Part (a)	`addClimb` (append)	**2 points**

Intent: *Create new* `ClimbInfo` *using data from parameters and append to* `climbList`

+1 Creates new `ClimbInfo` object using parametric data correctly

+1 Appends the created object to `climbList`
(*no bounds error and no destruction of existing data*)
(*point not awarded if inserted more than once*)

Part (b)	`addClimb` (alphabetical)	**6 points**

Intent: *Create new* `ClimbInfo` *object using data from parameters and insert into* `climbList`, *maintaining alphabetical order*

+1 Creates new `ClimbInfo` object(s), using parametric data correctly

+1 Compares `peakName` value with value retrieved from object in list (*must use* `getName`)

+1 Inserts object into list based on a comparison (other than equality) with object in list
(*point not awarded if inserted more than once*)

+1 Compares parametric data with all appropriate entries in `climbList` (*no bounds error*)

+1 Inserts new `ClimbInfo` object into `climbList` (*no destruction of existing data*)

+1 Inserts new `ClimbInfo` object into `climbList` once and only once in maintaining alphabetical order (*no destruction of existing data*)

Part (c)	analysis	**1 point**

Intent: *Analyze behavioral differences between* **append** *and* **alphabetical** *versions of* `addClimb`

+1 (i) NO (ii) YES Both must be answered correctly

Question-Specific Penalties

-1 (z) Attempts to return a value from `addClimb`

AP® COMPUTER SCIENCE A
2012 CANONICAL SOLUTIONS

Question 1: Climbing Club

Part (a):

```
public void addClimb(String peakName, int climbTime) {
   this.climbList.add(new ClimbInfo(peakName, climbTime));
}
```

Part (b):

```
public void addClimb(String peakName, int climbTime) {
   for (int i = 0; i < this.climbList.size(); i++) {
      if (peakName.compareTo(this.climbList.get(i).getName()) <= 0) {
         this.climbList.add(i, new ClimbInfo(peakName, climbTime));
         return;
      }
   }
   this.climbList.add(new ClimbInfo(peakName, climbTime));
}
```

Part (c):

NO

YES

These canonical solutions serve an expository role, depicting general approaches to solution. Each reflects only one instance from the infinite set of valid solutions. The solutions are presented in a coding style chosen to enhance readability and facilitate understanding.

Complete method addClimb below.

1Aa

```
/** Adds a new climb with name peakName and time climbTime to the list of climbs.
 *  @param peakName the name of the mountain peak climbed
 *  @param climbTime the number of minutes taken to complete the climb
 *  Postcondition: The new entry is at the end of climbList;
 *                 The order of the remaining entries is unchanged.
 */
public void addClimb(String peakName, int climbTime)
{
    climbList.add(New ClimbInfo(peakName, climbTime));
}
```

Part (b) begins on page 8.

Complete method addClimb below.

1Ab

```
/** Adds a new climb with name peakName and time climbTime to the list of climbs.
 *  Alphabetical order is determined by the compareTo method of the String class.
 *  @param peakName the name of the mountain peak climbed
 *  @param climbTime the number of minutes taken to complete the climb
 *  Precondition: entries in climbList are in alphabetical order by name.
 *  Postcondition: entries in climbList are in alphabetical order by name.
 */
public void addClimb(String peakName, int climbTime)
{
  int addPosition = climbList.size();
  for(int index = 0; index < climbList.size(); index++)
  {
    String nextName = climbList.get(index).getName();
    if(peakName.compareTo(nextName) < 0)
    {
      addPosition = index;
    }
  }
  climbList.add(addPosition, new ClimbInfo(peakName, climbTime));
}
```

Part (c) begins on page 10.

(c) The `ClimbingClub` method `distinctPeakNames` is intended to return the number of different names in `climbList`. For example, after the following code segment has completed execution, the value of the variable `numNames` would be 3.

1Ac

```
ClimbingClub hikerClub = new ClimbingClub();
hikerClub.addClimb("Monadnock", 274);
hikerClub.addClimb("Whiteface", 301);
hikerClub.addClimb("Algonquin", 225);
hikerClub.addClimb("Monadnock", 344);
int numNames = hikerClub.distinctPeakNames();
```

Consider the following implementation of method `distinctPeakNames`.

```
/** @return the number of distinct names in the list of climbs */
public int distinctPeakNames()
{
  if (climbList.size() == 0)
  {
    return 0;
  }

  ClimbInfo currInfo = climbList.get(0);
  String prevName = currInfo.getName();
  String currName = null;
  int numNames = 1;

  for (int k = 1; k < climbList.size(); k++)
  {
    currInfo = climbList.get(k);
    currName = currInfo.getName();
    if (prevName.compareTo(currName) != 0)
    {
       numNames++;
       prevName = currName;
    }
  }
  return numNames;
}
```

Assume that `addClimb` works as specified, regardless of what you wrote in parts (a) and (b).

(i) Does this implementation of the `distinctPeakNames` method work as intended when the `addClimb` method stores the `ClimbInfo` objects in the order they were added as described in part (a)?

Circle one of the answers below.

YES (NO)

(ii) Does this implementation of the `distinctPeakNames` method work as intended when the `addClimb` method stores the `ClimbInfo` objects in alphabetical order by name as described in part (b)?

Circle one of the answers below.

(YES) NO

Complete method `addClimb` below.

1Ba

```
/** Adds a new climb with name peakName and time climbTime to the list of climbs.
 *  @param peakName the name of the mountain peak climbed
 *  @param climbTime the number of minutes taken to complete the climb
 *  Postcondition: The new entry is at the end of climbList;
 *                 The order of the remaining entries is unchanged.
 */
public void addClimb(String peakName, int climbTime)
{
    ClimbInfo a = new ClimbInfo(peakName, climbTime);
    climbList.add(a);
}
```

Part (b) begins on page 8.

Complete method addClimb below.

1Bb

```
/** Adds a new climb with name peakName and time climbTime to the list of climbs.
 *  Alphabetical order is determined by the compareTo method of the String class.
 *  @param peakName the name of the mountain peak climbed
 *  @param climbTime the number of minutes taken to complete the climb
 *  Precondition: entries in climbList are in alphabetical order by name.
 *  Postcondition: entries in climbList are in alphabetical order by name.
 */
public void addClimb(String peakName, int climbTime)
{
    ClimbInfo b = new ClimbInfo(peakName, climbTime);
    for(int c = 0; c < climbList.size(); c++)
    {
        if( b.getName().compareTo(climbList.get(c).getName())
                                                    < 0)
        {
            climbList.add( c, b );
        }
    }
    if(b.getName().compareTo(climbList.get(climbList.size()-1)
                                                    > 0)
    {
        climbList.add( b);
    }
}
```

Part (c) begins on page 10.

(c) The ClimbingClub method distinctPeakNames is intended to return the number of different names in climbList. For example, after the following code segment has completed execution, the value of the variable numNames would be 3.

1Bc

```
ClimbingClub hikerClub = new ClimbingClub();
hikerClub.addClimb("Monadnock", 274);
hikerClub.addClimb("Whiteface", 301);
hikerClub.addClimb("Algonquin", 225);
hikerClub.addClimb("Monadnock", 344);
int numNames = hikerClub.distinctPeakNames();
```

Consider the following implementation of method distinctPeakNames.

```
/** @return the number of distinct names in the list of climbs */
public int distinctPeakNames()
{
  if (climbList.size() == 0)
  {
    return 0;
  }

  ClimbInfo currInfo = climbList.get(0);
  String prevName = currInfo.getName();
  String currName = null;
  int numNames = 1;

  for (int k = 1; k < climbList.size(); k++)
  {
    currInfo = climbList.get(k);
    currName = currInfo.getName();
    if (prevName.compareTo(currName) != 0)
    {
       numNames++;
       prevName = currName;
    }
  }
  return numNames;
}
```

Assume that addClimb works as specified, regardless of what you wrote in parts (a) and (b).

(i) Does this implementation of the distinctPeakNames method work as intended when the addClimb method stores the ClimbInfo objects in the order they were added as described in part (a)?

Circle one of the answers below.

YES NO

(ii) Does this implementation of the distinctPeakNames method work as intended when the addClimb method stores the ClimbInfo objects in alphabetical order by name as described in part (b)?

Circle one of the answers below.

YES NO

Complete method addClimb below.

1Ca

```
/** Adds a new climb with name peakName and time climbTime to the list of climbs.
 *  @param peakName the name of the mountain peak climbed
 *  @param climbTime the number of minutes taken to complete the climb
 *  Postcondition: The new entry is at the end of climbList;
 *                 The order of the remaining entries is unchanged.
 */
public void addClimb(String peakName, int climbTime)
{
   ClimbInfo c = new ClimbInfo(string peakName, int climbTime);
   climbList.add(climbList.size()-1, c);
}
```

Part (b) begins on page 8.

Complete method addClimb below.

1Cb

```
/** Adds a new climb with name peakName and time climbTime to the list of climbs.
 *  Alphabetical order is determined by the compareTo method of the String class.
 *  @param peakName the name of the mountain peak climbed
 *  @param climbTime the number of minutes taken to complete the climb
 *  Precondition: entries in climbList are in alphabetical order by name.
 *  Postcondition: entries in climbList are in alphabetical order by name.
 */
public void addClimb(String peakName, int climbTime)
{
    ClimbInfo c = new ClimbInfo( String peakName, int climbTime);

    for(int i = 0; i< climbList.size(); i++)
    {
        String name = climbList.get(i).getName();

        int compare = name.compareTo(c.getName());

        if( compare < 0)
        {
          climbList.add(climbList.get(i+1), c);
        }
    }

}
```

Part (c) begins on page 10.

(c) The `ClimbingClub` method `distinctPeakNames` is intended to return the number of different names in `climbList`. For example, after the following code segment has completed execution, the value of the variable `numNames` would be 3.

1Cc

```
ClimbingClub hikerClub = new ClimbingClub();
hikerClub.addClimb("Monadnock", 274);
hikerClub.addClimb("Whiteface", 301);
hikerClub.addClimb("Algonquin", 225);
hikerClub.addClimb("Monadnock", 344);
int numNames = hikerClub.distinctPeakNames();
```

Consider the following implementation of method `distinctPeakNames`.

```
/** @return the number of distinct names in the list of climbs */
public int distinctPeakNames()
{
  if (climbList.size() == 0)
  {
    return 0;
  }

  ClimbInfo currInfo = climbList.get(0);
  String prevName = currInfo.getName();
  String currName = null;
  int numNames = 1;

  for (int k = 1; k < climbList.size(); k++)
  {
    currInfo = climbList.get(k);
    currName = currInfo.getName();
    if (prevName.compareTo(currName) != 0)
    {
       numNames++;
       prevName = currName;
    }
  }
  return numNames;
}
```

Assume that `addClimb` works as specified, regardless of what you wrote in parts (a) and (b).

(i) Does this implementation of the `distinctPeakNames` method work as intended when the `addClimb` method stores the `ClimbInfo` objects in the order they were added as described in part (a)?

Circle one of the answers below.

(YES) NO

(ii) Does this implementation of the `distinctPeakNames` method work as intended when the `addClimb` method stores the `ClimbInfo` objects in alphabetical order by name as described in part (b)?

Circle one of the answers below.

YES (NO)

AP® COMPUTER SCIENCE A
2012 SCORING COMMENTARY

Question 1

Overview

This question involved object construction, list access and modification, string comparison, and implications of design choices. Students were asked to implement the method `addClimb` using two different specifications of its behavior. In both cases, the main action of `addClimb` was to use its parameters to create a new object of type `ClimbInfo` and add that object to the instance variable `climbList`. In part (a) students were required to implement `addClimb` such that the new object was simply appended to `climbList`. This could be accomplished by invoking the one-parameter `add` method of the `List` interface. In part (b) students were required to implement `addClimb` such that the new object was inserted into `climbList` so as to maintain alphabetical order of the elements. This could be accomplished by searching the list to determine the location at which the element should be inserted and invoking the two-parameter `add` method of the `List` interface. Part (c) provided an implementation of the method `distinctPeakNames` that traverses `climbList` and attempts to determine the number of distinct values of names of the peaks. Students were asked whether that implementation of `distinctPeakNames` works as intended when `addClimb` stored the objects as in part (a) and in part (b).

Sample: 1A
Score: 8

In part (a) the student correctly creates a new `ClimbInfo` object using the method's parameters as the arguments to the `ClimbInfo` constructor. The new object is then appended to the list via the one-argument version of the `add` method. The capitalization of "`New`" (which should be "`new`") is ignored in scoring, because unambiguous spelling/case discrepancies are not penalized under the General Scoring Guidelines. The student earned both points in part (a).

In part (b) the student attempts to determine where a new `ClimbInfo` object belongs in `climbList`, and then creates and inserts it. The student uses a `for` loop to compare the `peakName` parameter with the peak name of each object in the list. The `compareTo` method for `String` objects is used to determine if the new object belongs before the current object in the list. If the new object does belong before the current object, that position is recorded in a local variable (`addPosition`), which was initialized to the size of the list.

After the loop, the new object is added via the two-argument `add` method, in which the first parameter indicates the insert position in the list. For an empty list, the size is 0, and thus the object is correctly inserted at index 0. If the new object belongs at the end of the list, the object is correctly appended. However, if the new object does not belong at the end of the list, there is a problem. Once the loop finds an object in the list that belongs *after* the new object, it correctly records that position, but the loop continues. Because all subsequent objects in the list also belong after the new object, the local variable gets updated for each of these additional objects. This results in the new object being inserted immediately before the last object in the list. The student earned 5 points in part (b).

In part (c) the student correctly indicates that the `distinctPeakNames` implementation works only if the list is ordered. The `addClimb` specification for part (a) does not do this, whereas the part (b) specification does guarantee an ordered list. The student earned the point for part (c).

Question 1 (continued)

Sample: 1B
Score: 6

In part (a) the student creates a new `ClimbInfo` object using the method's parameters as the arguments to the `ClimbInfo` constructor. The new object is then appended to the list using the one-argument version of the `add` method. The student earned both points in part (a).

In part (b) the student creates a new `ClimbInfo` object, attempts to determine where it belongs in `climbList`, and inserts it into the list. The response uses a `for` loop to compare the `peakName` parameter with the peak name of each object in the list. The `compareTo` method for `String` objects is used to determine if the new object belongs before the current object in the list. If the new object does belong before the current object, the new object is inserted. However, because the loop continues, the next pass through the loop will compare the new object to the same object it was compared to in the previous iteration, resulting in the new object being inserted multiple times. This process will continue indefinitely. Neither the "Inserts object into list based on a comparison" point nor the "Inserts new `ClimbInfo` object into `climbList` once and only once" point can be earned when there are multiple inserts.

After the loop, the student checks to see if the new object belongs at the end of the list, and if so, appends it. However, if the list is empty, this check will cause an exception (accessing the object in position -1) and the new object will not get inserted. Neither the "Inserts object into `ClimbList`" point nor the "Inserts object into `climbList` once and only once" point can be earned when there is a missing insert. The student earned 3 points for part (b).

In part (c) the student correctly indicates that the `distinctPeakNames` implementation works only if the list is ordered. The `addClimb` specification for part (a) does not do this, whereas the part (b) specification does guarantee an ordered list. The student earned the point for part (c).

Sample: 1C
Score: 1

In part (a) the student incorrectly creates a new `ClimbInfo` object, because the parameter list includes the types of arguments, as a formal parameter list would. The object is then added to the list, but not after all previous objects. The first argument to the `add` method indicates the position in the list. Because `climbList.size()- 1` is used as that argument, the new object is inserted before the last object, making this new object the penultimate object in the list. Additionally, if the list is empty at the beginning of the method, there will be an exception when it tries to add at index -1. No points were earned in part (a).

In part (b) the student attempts to create a new `ClimbInfo` object, determine where it belongs in `climbList`, and then insert it. The student incorrectly creates a new `ClimbInfo` object, because the parameter list includes the types of arguments, as a formal parameter list would.

The student uses a `for` loop to compare the peak name of each object in the list with the `peakName` parameter. The `compareTo` method for `String` objects is used in an attempt to determine if the new object belongs before the current object in the list. However, the `compare < 0` test should be `compare > 0`. When the `if` test is true, an `add` method is called, but the arguments to the `add` method do not match either of the two `List add` methods. Both of the arguments are objects, so it is

Question 1 (continued)

unclear which object is intended to be inserted into the list. Also, because the first argument accesses the object at index `i+1`, an exception will occur when `i` equals `climbList.size()-1`. If the `add` method had the correct parameters, the loop would continue after the object is inserted into the list, and additional calls to the `add` method could occur. This would result in the new object being inserted more than once. Neither the "Inserts object based on comparison" point nor the "Inserts object into `climbList` once and only once" point can be earned when there are multiple inserts.

If the list was empty when the method began, the `for` loop will not execute, and no `add` will occur. Neither the "Inserts object into `ClimbList`" point nor the "Inserts object into `climbList` once and only once" point can be earned when there is a missing insert. The student earned 1 point for part (b).

In part (c) the `distinctPeakNames` implementation works only if the list is ordered. The student incorrectly indicates that the `addClimb` specification for part (a) guarantees this, whereas the part (b) specification does not. In fact, the part (b) specification is the one that enables `distinctPeakNames` to work correctly, and the part (a) specification does not. No points were earned in part (c).