AP® COMPUTER SCIENCE A
2006 SCORING GUIDELINES

Question 3: Customer List

Part A: `compareCustomer` **3 points**

+1 1/2 perform comparison
- **+1/2** attempt (must call `OBJ1.compareTo(OBJ2)`)
- **+1/2** correctly access and compare names
- **+1/2** correctly access and compare IDs

+1/2 return 0 if and only if `this = other`
+1/2 return positive if and only if `this > other`
+1/2 return negative if and only if `this < other`

Part B: `prefixMerge` **6 points**

+1/2 initialize unique variables to index fronts of arrays

+1 1/2 loop over arrays to fill result
- **+1/2** attempt (must reference `list1` and `list2` inside loop)
- **+1** correct (lose this if add too few or too many `Customer` elements)

+1 1/2 compare array fronts (in context of loop)
- **+1/2** attempt (must call `compareCustomer` on array elements)
- **+1** correctly compare front `Customer` elements

+1 1/2 duplicate entries
- **+1/2** check if duplicate entries found
- **+1/2** if duplicates, copy only one to `result` (without use of additional structure)
- **+1/2** update indices into both arrays (`list1` and `list2`)

+1 nonduplicate entries
- **+1/2** copy only smallest entry to `result` (without use of additional structure)
- **+1/2** update index into that array only (`list1` or `list2`)

Note: Solution may use constants as returned from part A.

Usage: -1/2 `compareTo` instead of `compareCustomer` for `Customer` objects

AP® COMPUTER SCIENCE A/AB
2006 GENERAL USAGE

Most common usage errors are addressed specifically in rubrics with points deducted in a manner other than indicated on this sheet. The rubric takes precedence.

Usage points can only be deducted if the part where it occurs has earned credit.

A usage error that occurs once when the same usage is correct two or more times can be regarded as an oversight and not penalized. If the usage error is the only instance, one of two, or occurs two or more times, then it should be penalized.

A particular usage error should be penalized only once in a problem, even if it occurs on different parts of a problem.

Nonpenalized Errors

spelling/case discrepancies*

local variable not declared when any other variables are declared in some part

default constructor called without parens; for example, `new Fish;`

use keyword as identifier

`[r,c]`, `(r)(c)` or `(r,c)` instead of `[r][c]`

= instead of == (and vice versa)

`length`/`size` confusion for array, `String`, and `ArrayList`, with or without `()`

`private` qualifier on local variable

extraneous code with no side-effect, for example a check for precondition

common mathematical symbols for operators (x • ÷ ≤ ≥ < > ≠)

missing { } where indentation clearly conveys intent

missing `( )` on method call or around `if/while` conditions

missing `;`s

missing "new" for constructor call once, when others are present in some part

missing downcast from collection

missing `int` cast when needed

missing `public` on class or constructor header

Minor Errors (1/2 point)

confused identifier (e.g., `len` for `length` or `left()` for `getLeft()`)

no local variables declared

`new` never used for constructor calls

`void` method or constructor returns a value

modifying a constant (`final`)

use `equals` or `compareTo` method on primitives, for example
`int x; …x.equals(val)`

`[]` – `get` confusion if access not tested in rubric

assignment dyslexia, for example, `x + 3 = y;` for `y = x + 3;`

`super(method())` instead of `super.method()`

formal parameter syntax (with type) in method call, e.g., `a = method(int x)`

missing `public` from method header when required

"false"/"true" or 0/1 for boolean values

"null" for `null`

Major Errors (1 point)

extraneous code which causes side-effect, for example, information written to output

use interface or class name instead of variable identifier, for example `Simulation.step()` instead of `sim.step()`

`aMethod(obj)` instead of `obj.aMethod()`

use of object reference that is incorrect, for example, use of `f.move()` inside method of `Fish` class

use private data or method when not accessible

destruction of data structure (e.g., by using root reference to a `TreeNode` for traversal of the tree)

use class name in place of `super` either in constructor or in method call

**Note: Spelling and case discrepancies for identifiers fall under the "nonpenalized" category as long as the correction can be unambiguously inferred from context. For example, "Queu" instead of "Queue". Likewise, if a student declares "Fish fish;", then uses Fish.move() instead of fish.move(), the context allows for the reader to assume the object instead of the class.*

A3 A,

Complete method compareCustomer below.

```
// returns 0 when this customer is equal to other;
//   a positive integer when this customer is greater than other;
//   a negative integer when this customer is less than other
public int compareCustomer(Customer other)
{
    int c = getName().compareTo(other.getName());
    if (c!=0){
        return c;
    } else {
        int rd = other.getID();
        if (getID() > rd) return 1;
        else if (getID() < rd) return -1;
        else return 0;
    }
}
```

Part (b) begins on page 16.

A3 A2

Complete method prefixMerge below.

```
// fills result with customers merged from the
// beginning of list1 and list2;
// result contains no duplicates and is sorted in
// ascending order by customer
// precondition:  result.length > 0;
//                list1.length >= result.length;
//                list1 contains no duplicates;
//                list2.length >= result.length;
//                list2 contains no duplicates;
//                list1 and list2 are sorted in
//                ascending order by customer
// postcondition: list1, list2 are not modified
public static void prefixMerge(Customer[] list1,
                               Customer[] list2,
                               Customer[] result)
{
    int index1, index2;
    index1=0;
    index2=0;
    for (int i=0; i<result.length; i++){
        int c = list1[index1].compareCustomer(list2[index2]);
        if (c<0){
            result[i] = list1[index1];
            index1++;
        } else if (c>0){
            result[i] = list2[index2];
            index2++
        } else {  // c==0
            result[i] = list1[index1];
            index1++;
            index2++;
        }
    }
}
```

GO ON TO THE NEXT PAGE.

A3 B,

Complete method compareCustomer below.

```
// returns 0 when this customer is equal to other;
//   a positive integer when this customer is greater than other;
//   a negative integer when this customer is less than other
public int compareCustomer(Customer other)
{
        if ( (this.getName()).compareTo(other.getName()) > 0 )
            return -1 ;
        else if ( (this.getName()).compareTo(other.getName()) == 0 )
        {
            if ( this.getID() > other.getID() )
                return 1 ;
            else if ( this.getID() < other.getID() )
                return -1 ;
            else
                return 0 ;
        }
        else
            return 1 ;
}
```

Part (b) begins on page 16.

A3 B2

Complete method prefixMerge below.

```
// fills result with customers merged from the
// beginning of list1 and list2;
// result contains no duplicates and is sorted in
// ascending order by customer
// precondition:  result.length > 0;
//                list1.length >= result.length;
//                list1 contains no duplicates;
//                list2.length >= result.length;
//                list2 contains no duplicates;
//                list1 and list2 are sorted in
//                ascending order by customer
// postcondition: list1, list2 are not modified
public static void prefixMerge(Customer[] list1,
                               Customer[] list2,
                               Customer[] result)
{
        for (int i = 0; i < list1.length; i++)
        {
            for (int j = 0; j < list2.length; j++)
            {
                for (int k = 0; k < result.length; k++)
                {
                    if (list1[i].compareCustomer(list2[j]) == 0)
                        result[k] = list1[i];
                    else if (list1[i].compareCustomer(list2[j]) < 0)
                        result[k] = list1[i];
                    else
                        result[k] = list2[j];
                }
            }
        }
}
```

A3 C1

Complete method compareCustomer below.

```
// returns 0 when this customer is equal to other;
//   a positive integer when this customer is greater than other;
//   a negative integer when this customer is less than other
public int compareCustomer(Customer other)
{
   if (getName() == other.getName())
      return 0;
   else if( getID() > other.getID())
      return 1;
   else
      return -1;
}
```

Part (b) begins on page 16.

A3 C2

Complete method prefixMerge below.

```
// fills result with customers merged from the
// beginning of list1 and list2;
// result contains no duplicates and is sorted in
// ascending order by customer
// precondition:  result.length > 0;
//                list1.length >= result.length;
//                list1 contains no duplicates;
//                list2.length >= result.length;
//                list2 contains no duplicates;
//                list1 and list2 are sorted in
//                ascending order by customer
// postcondition: list1, list2 are not modified
public static void prefixMerge(Customer[] list1,
                               Customer[] list2,
                               Customer[] result)
{
  for(int x=0; x<= result.length; x++)
  {
    if(list1[x].compareCustomer(list2[x])==0)
      result.add(list1[x]);
    else if(list1[x].compareCustomer(list2[x])>0)
      result.add(list1[x]);
    else
      result.add(list2[x]);
  }
}
```

AP® COMPUTER SCIENCE A
2006 SCORING COMMENTARY

Question 3

Overview

This question focused on abstraction, array traversal, and the application of basic algorithms. In part (a) students were given a class to represent customers. The `Customer` class had accessor methods for getting a customer's name and ID, and the students were required to complete the `compareCustomer` method that compared two customers. This involved calling the name and ID accessors on both customers, comparing names (using the `String compareTo` method), and also comparing IDs in the case of identical names. In part (b) students were required to complete a method that took two sorted arrays of `Customers` and merged them into a single array of fixed length. This involved maintaining indexes to the front of the arrays, repeatedly comparing customers from the fronts (using the `compareCustomer` method from part (a)), and copying the "smaller" customer to the merged array.

Sample: A3A
Score: 9

In part (a) the student correctly accesses and compares names. When they are different, the method correctly returns the result of the `compareTo` method of the `String` class. When names are the same, the IDs are correctly accessed and compared, and an acceptable value is returned in all cases.

Part (b) is completely correct. The student uses a loop to fill `result`, using independent indices to compare elements in `list1` and `list2`. Only the smaller entry is copied when the compared elements are not equal, and the case of duplicate entries is handled correctly.

Sample: A3B
Score: 6

In part (a) the student correctly accesses and compares names, but the logic is incorrect when names are not equal, losing those two ½ points. The method returns the correct value when the `Customer` objects are equal.

In part (b) the index variable `j` is changed automatically without regard to merging logic. This lost the 1 point for a correct comparison to the fronts of the two arrays, the ½ point for updating array indices in the case of duplicates, and the ½ point for updating a single array index when there are not duplicates. All other points were earned in this part.

Sample: A3C
Score: 2

In part (a) only the IDs are compared correctly, earning a ½ point. No other points were awarded because there was no call to the method `compareTo`, and the return logic was incorrect.

Part (b) earned 1½ points: the ½ point for attempting a loop, the ½ point for attempting to compare elements in `list1` and `list2`, and the ½ point for checking for duplicate elements. The index variable `x` is changed automatically without regard to merging logic. The response lost the 1 point for a correct comparison to the fronts of the two arrays, the ½ point for updating array indices in the case of duplicates, and the ½ point for updating a single array index when there are not duplicates. It also lost the initialization ½ point because there are not multiple index variables. Finally, the student lost the 1 point for loop correctness because of an incorrect loop bound.