CollegeBoard
Advanced Placement Program

AP® Computer Science A
2004 Free-Response Questions

The materials included in these files are intended for noncommercial use by AP teachers for course and exam preparation; permission for any other use must be sought from the Advanced Placement Program®. Teachers may reproduce them, in whole or in part, in limited quantities, for face-to-face teaching purposes but may not mass distribute the materials, electronically or otherwise. This permission does not apply to any third-party copyrights contained herein. These materials and any copies made of them may not be resold, and the copyright notices must be retained as they appear here.

The College Board is a not-for-profit membership association whose mission is to connect students to college success and opportunity. Founded in 1900, the association is composed of more than 4,500 schools, colleges, universities, and other educational organizations. Each year, the College Board serves over three million students and their parents, 23,000 high schools, and 3,500 colleges through major programs and services in college admissions, guidance, assessment, financial aid, enrollment, and teaching and learning. Among its best-known programs are the SAT®, the PSAT/NMSQT®, and the Advanced Placement Program® (AP®). The College Board is committed to the principles of excellence and equity, and that commitment is embodied in all of its programs, services, activities, and concerns.

For further information, visit www.collegeboard.com

Copyright © 2004 College Entrance Examination Board. All rights reserved. College Board, Advanced Placement Program, AP, AP Central, AP Vertical Teams, APCD, Pacesetter, Pre-AP, SAT, Student Search Service, and the acorn logo are registered trademarks of the College Entrance Examination Board. PSAT/NMSQT is a registered trademark jointly owned by the College Entrance Examination Board and the National Merit Scholarship Corporation. Educational Testing Service and ETS are registered trademarks of Educational Testing Service. Other products and services may be trademarks of their respective owners.

For the College Board's online home for AP professionals, visit AP Central at apcentral.collegeboard.com.

COMPUTER SCIENCE A
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.
- 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.

Copyright © 2004 by College Entrance Examination Board. All rights reserved.
Visit apcentral.collegeboard.com (for AP professionals) and www.collegeboard.com/apstudents (for AP students and parents).

1. The following class `WordList` is designed to store and manipulate a list of words. The incomplete class declaration is shown below. You will be asked to implement two methods.

```
public class WordList
{
  private ArrayList myList;  // contains Strings made up of letters

  // postcondition: returns the number of words in this WordList that
  //                are exactly len letters long
  public int numWordsOfLength(int len)
  {  /* to be implemented in part (a) */  }

  // postcondition: all words that are exactly len letters long
  //                have been removed from this WordList, with the
  //                order of the remaining words unchanged
  public void removeWordsOfLength(int len)
  {  /* to be implemented in part (b) */  }

  // ... constructor and other methods not shown
}
```

(a) Write the `WordList` method `numWordsOfLength`. Method `numWordsOfLength` returns the number of words in the `WordList` that are exactly `len` letters long. For example, assume that the instance variable `myList` of the `WordList animals` contains the following.

```
["cat", "mouse", "frog", "dog", "dog"]
```

The table below shows several sample calls to `numWordsOfLength`.

Call	Result returned by call
`animals.numWordsOfLength(4)`	1
`animals.numWordsOfLength(3)`	3
`animals.numWordsOfLength(2)`	0

Complete method `numWordsOfLength` below.

```
  // postcondition: returns the number of words in this WordList that
  //                are exactly len letters long
  public int numWordsOfLength(int len)
```

Copyright © 2004 by College Entrance Examination Board. All rights reserved.
Visit apcentral.collegeboard.com (for AP professionals) and www.collegeboard.com/apstudents (for AP students and parents).

(b) Write the `WordList` method `removeWordsOfLength`. Method `removeWordsOfLength` removes all words from the `WordList` that are exactly `len` letters long, leaving the order of the remaining words unchanged. For example, assume that the instance variable `myList` of the `WordList` `animals` contains the following.

```
["cat", "mouse", "frog", "dog", "dog"]
```

The table below shows a <u>sequence</u> of calls to the `removeWordsOfLength` method.

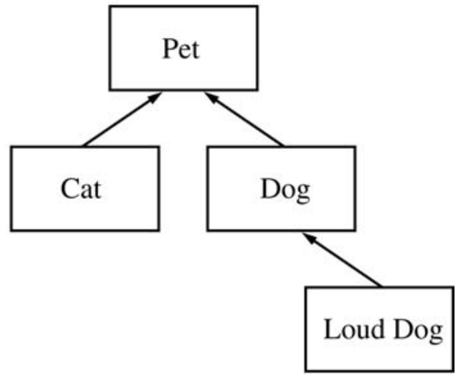
Call	`myList` after the call
`animals.removeWordsOfLength(4);`	`["cat", "mouse", "dog", "dog"]`
`animals.removeWordsOfLength(3);`	`["mouse"]`
`animals.removeWordsOfLength(2);`	`["mouse"]`

Complete method `removeWordsOfLength` below.

```
// postcondition: all words that are exactly len letters long
//                have been removed from this WordList, with the
//                order of the remaining words unchanged
public void removeWordsOfLength(int len)
```

Copyright © 2004 by College Entrance Examination Board. All rights reserved.
Visit apcentral.collegeboard.com (for AP professionals) and www.collegeboard.com/apstudents (for AP students and parents).

2. Consider the hierarchy of classes shown in the following diagram.



Note that a `Cat` "is-a" `Pet`, a `Dog` "is-a" `Pet`, and a `LoudDog` "is-a" `Dog`.

The class `Pet` is specified as an abstract class as shown in the following declaration. Each `Pet` has a name that is specified when it is constructed.

```
public abstract class Pet
{
  private String myName;

  public Pet(String name)
  {  myName = name;  }

  public String getName()
  {  return myName;  }

  public abstract String speak();
}
```

The subclass `Dog` has the partial class declaration shown below.

```
public class Dog extends Pet
{
  public Dog(String name)
  {  /* implementation not shown */  }

  public String speak()
  {  /* implementation not shown */  }
}
```

Copyright © 2004 by College Entrance Examination Board. All rights reserved.
Visit apcentral.collegeboard.com (for AP professionals) and www.collegeboard.com/apstudents (for AP students and parents).

(a) Given the class hierarchy shown above, write a complete class declaration for the class `Cat`, including implementations of its constructor and method(s). The `Cat` method `speak` returns `"meow"` when it is invoked.

(b) Assume that class `Dog` has been declared as shown at the beginning of the question. If the `String` *dog-sound* is returned by the `Dog` method `speak`, then the `LoudDog` method `speak` returns a `String` containing *dog-sound* repeated two times.

Given the class hierarchy shown previously, write a complete class declaration for the class `LoudDog`, including implementations of its constructor and method(s).

(c) Consider the following partial declaration of class `Kennel`.

```
public class Kennel
{
  private ArrayList petList;    // all elements are references
                                // to Pet objects

  // postcondition: for each Pet in the kennel, its name followed
  //                by the result of a call to its speak method
  //                has been printed, one line per Pet
  public void allSpeak()
  {  /* to be implemented in this part */  }

  // ... constructor and other methods not shown
}
```

Write the `Kennel` method `allSpeak`. For each `Pet` in the kennel, `allSpeak` prints a line with the name of the `Pet` followed by the result of a call to its `speak` method.

In writing `allSpeak`, you may use any of the methods defined for any of the classes specified for this problem. Assume that these methods work as specified, regardless of what you wrote in parts (a) and (b). Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `allSpeak` below.

```
  // postcondition: for each Pet in the kennel, its name followed
  //                by the result of a call to its speak method
  //                has been printed, one line per Pet
  public void allSpeak()
```

Copyright © 2004 by College Entrance Examination Board. All rights reserved.
Visit apcentral.collegeboard.com (for AP professionals) and www.collegeboard.com/apstudents (for AP students and parents).

4. The PR2004 is a robot that automatically gathers toys and other items scattered in a tiled hallway. A tiled hallway has a wall at each end and consists of a single row of tiles, each with some number of items to be gathered.

The PR2004 robot is initialized with a starting position and an array that contains the number of items on each tile. Initially the robot is facing right, meaning that it is facing toward higher-numbered tiles.

The PR2004 robot makes a sequence of moves until there are no items remaining on any tile. A move is defined as follows.

1. If there are any items on the current tile, then one item is removed.
2. If there are more items on the current tile, then the robot remains on the current tile facing the same direction.
3. If there are no more items on the current tile
   a) if the robot can move forward, it advances to the next tile in the direction that it is facing;
   b) otherwise, if the robot cannot move forward, it reverses direction and does not change position.

In the following example, the position and direction of the robot are indicated by "<" or ">" and the entries in the diagram indicate the number of items to be gathered on each tile. There are four tiles in this hallway. The starting state of the robot is illustrated in the following diagram.

Tile number:		0	1	2	3	
Number of items:	left wall →	**1**	**1**	**2**	**2**	← right wall
Robot position:			>			

The following sequence shows the configuration of the hallway and the robot after each move.

After move 1

0	1	2	3
1	**0**	**2**	**2**
		>	

After move 2

0	1	2	3
1	**0**	**1**	**2**
		>	

After move 3

0	1	2	3
1	**0**	**0**	**2**
			>

After move 4

0	1	2	3
1	**0**	**0**	**1**
			>

After move 5

0	1	2	3
1	**0**	**0**	**0**
			<

After move 6

0	1	2	3
1	**0**	**0**	**0**
		<	

After move 7

0	1	2	3
1	**0**	**0**	**0**
	<		

After move 8

0	1	2	3
1	**0**	**0**	**0**
<			

After move 9

0	1	2	3
0	**0**	**0**	**0**
>			

After nine moves, the robot stops because the hall is clear.

Copyright © 2004 by College Entrance Examination Board. All rights reserved.
Visit apcentral.collegeboard.com (for AP professionals) and www.collegeboard.com/apstudents (for AP students and parents).

The PR2004 is modeled by the class `Robot` as shown in the following declaration.

```
public class Robot
{
  private int[] hall;
  private int pos;                // current position(tile number) of Robot
  private boolean facingRight;    // true means this Robot is facing right

  // constructor not shown

  // postcondition: returns true if this Robot has a wall immediately in
  //                front of it, so that it cannot move forward;
  //                otherwise, returns false
  private boolean forwardMoveBlocked()
  {  /* to be implemented in part (a) */  }

  // postcondition: one move has been made according to the
  //                specifications above and the state of this
  //                Robot has been updated
  private void move()
  {  /* to be implemented in part (b) */  }

  // postcondition: no more items remain in the hallway;
  //                returns the number of moves made
  public int clearHall()
  {  /* to be implemented in part (c) */  }

  // postcondition: returns true if the hallway contains no items;
  //                otherwise, returns false
  private boolean hallIsClear()
  {  /* implementation not shown */  }
}
```

In the `Robot` class, the number of items on each tile in the hall is stored in the corresponding entry in the array `hall`. The current position is stored in the instance variable `pos`. The `boolean` instance variable `facingRight` is `true` if the `Robot` is facing to the right and is `false` otherwise.

(a) Write the `Robot` method `forwardMoveBlocked`. Method `forwardMoveBlocked` returns `true` if the robot has a wall immediately in front of it, so that it cannot move forward. Otherwise, `forwardMoveBlocked` returns `false`.

Complete method `forwardMoveBlocked` below.

```
// postcondition: returns true if this Robot has a wall immediately in
//                front of it, so that it cannot move forward;
//                otherwise, returns false
private boolean forwardMoveBlocked()
```

Copyright © 2004 by College Entrance Examination Board. All rights reserved.
Visit apcentral.collegeboard.com (for AP professionals) and www.collegeboard.com/apstudents (for AP students and parents).

(b) Write the `Robot` method `move`. Method `move` has the robot carry out one move as specified at the beginning of the question. The specification for a move is repeated here for your convenience.

1. If there are any items on the current tile, then one item is removed.
2. If there are more items on the current tile, then the robot remains on the current tile facing the same direction.
3. If there are no more items on the current tile
   a) if the robot can move forward, it advances to the next tile in the direction that it is facing;
   b) otherwise, if the robot cannot move forward, it reverses direction and does not change position.

In writing `move`, you may use any of the other methods in the `Robot` class. Assume these methods work as specified, regardless of what you wrote in part (a). Solutions that reimplement the functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `move` below.

```
// postcondition: one move has been made according to the
//                specifications above and the state of this
//                Robot has been updated
private void move()
```

(c) Write the `Robot` method `clearHall`. Method `clearHall` clears the hallway, repeatedly having this robot make a move until the hallway has no items, and returns the number of moves made.

In the example at the beginning of this problem, `clearHall` would take the robot through the moves shown and return 9, leaving the robot in the state shown in the final diagram.

In writing `clearHall`, you may use any of the other methods in the `Robot` class. Assume these methods work as specified, regardless of what you wrote in parts (a) and (b). Solutions that reimplement the functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `clearHall` below.

```
// postcondition: no more items remain in the hallway;
//                returns the number of moves made
public int clearHall()
```

END OF EXAMINATION

Copyright © 2004 by College Entrance Examination Board. All rights reserved.
Visit apcentral.collegeboard.com (for AP professionals) and www.collegeboard.com/apstudents (for AP students and parents).

CollegeBoard
Advanced Placement Program

AP® Computer Science A 2004 Scoring Guidelines

The materials included in these files are intended for noncommercial use by AP teachers for course and exam preparation; permission for any other use must be sought from the Advanced Placement Program®. Teachers may reproduce them, in whole or in part, in limited quantities, for face-to-face teaching purposes but may not mass distribute the materials, electronically or otherwise. This permission does not apply to any third-party copyrights contained herein. These materials and any copies made of them may not be resold, and the copyright notices must be retained as they appear here.

The College Board is a not-for-profit membership association whose mission is to connect students to college success and opportunity. Founded in 1900, the association is composed of more than 4,500 schools, colleges, universities, and other educational organizations. Each year, the College Board serves over three million students and their parents, 23,000 high schools, and 3,500 colleges through major programs and services in college admissions, guidance, assessment, financial aid, enrollment, and teaching and learning. Among its best-known programs are the SAT®, the PSAT/NMSQT®, and the Advanced Placement Program® (AP®). The College Board is committed to the principles of excellence and equity, and that commitment is embodied in all of its programs, services, activities, and concerns.

For further information, visit www.collegeboard.com

Copyright © 2004 College Entrance Examination Board. All rights reserved. College Board, Advanced Placement Program, AP, AP Central, AP Vertical Teams, APCD, Pacesetter, Pre-AP, SAT, Student Search Service, and the acorn logo are registered trademarks of the College Entrance Examination Board. PSAT/NMSQT is a registered trademark of the College Entrance Examination Board and National Merit Scholarship Corporation. Educational Testing Service and ETS are registered trademarks of Educational Testing Service. Other products and services may be trademarks of their respective owners.

For the College Board's online home for AP professionals, visit AP Central at apcentral.collegeboard.com.

Question 1

Part A:	`numWordsOfLength`	**4 pts**

+1/2 declare and initialize count to zero (could be an empty list, length = 0)
(must show evidence that variable is used for counting or returned)

+1 loop over `myList`
- **+1/2** attempt (must reference `myList` in body)
- **+1/2** correct

+1/2 get `String` from `myList` (no deduction for missing downcast but local must be `String`)
(lose this if array syntax used)

+1 check length of `String`
- **+1/2** attempt (must be in context of loop)
- **+1/2** correct (array syntax is OK)

+1/2 increment count (must be within context of length check)
(lose this if count does not accumulate)

+1/2 return correct count (after loop is completed)

Part B:	`removeWordsOfLength`	**5 pts**

+2 loop over `myList`
- **+1** attempt (must reference `myList` in body)
- **+1** correct (must have attempt at removal, must not skip items)

+1 get `String` from `myList` (no deduction for missing downcast, but local must be `String`)
- **+1/2** attempt (must be in context of loop, array syntax is OK)
- **+1/2** correct (no array syntax)

+1 check length of `String`
- **+1/2** attempt (must be in context of loop)
- **+1/2** correct (array syntax is OK)

+1 remove
- **+1/2** attempt (must call `remove`, must refer to `myList` or an index of an element in `myList`)
- **+1/2** correct (no array syntax)

Usage:
- **-1/2** for `WordList` instead of `myList`
- **-1/2** for returning a value in part B
- **-1** for using `this` instead of `myList`, can lose in part A and again in part B (for max of -2)

Copyright © 2004 by College Entrance Examination Board. All rights reserved.
Visit apcentral.collegeboard.com (for AP professionals) and www.collegeboard.com/apstudents (for AP students and parents).

Question 2

Part A:	`class Cat`	**2 pts**

+1/2 `public class Cat extends Pet`

+1/2 Constructor correct (must call `super`)

+1 `speak` method
- **+1/2** attempt (method header matches abstract method, OK if `abstract` left in)
- **+1/2** correct

Part B:	`class LoudDog`	**3 pts**

+1/2 `public class LoudDog extends Dog`

+1 Constructor correct (must call `super`)

+1 1/2 `speak` method
- **+1** attempt (calls `super.speak()` *and* method header matches abstract method, OK if `abstract` left in)
- **+1/2** correct value returned

Part C:	Kennel - `allSpeak`	**4 pts**

+1 loop over `petList`
- **+1/2** attempt
- **+1/2** correct (must access `petList`)

+1 1/2 get pet from `petList` (no deduction for missing downcast from `petList`)
- **+1/2** attempt
- **+1** correct (local variable must be type `Pet`)

+1 1/2 print `p.getName()` and `p.speak()` for pet `p` (local variable not necessary)
- **+1/2** attempt (must have `xxx.getName()` or `xxx.speak()`, for some xxx)
- **+1** correct

Note: if done in-line with no local, no deduction for missing downcast.

Usage:
- **-1/2** `public` instance variable
- **-1** parent class name instead of `super`
- **-1/2** `getName` is overridden (other than `super.getName`) in part (a) and/or part (b)

(No deduction for other additional methods or constructors.)

Copyright © 2004 by College Entrance Examination Board. All rights reserved.
Visit apcentral.collegeboard.com (for AP professionals) and www.collegeboard.com/apstudents (for AP students and parents).

Question 4

Part A:	`forwardMoveBlocked`	**1 pt**

+1 return boolean
- **+1/2** check a dir/pos pair
- **+1/2** correct

Part B:	`move`	**5 pts**

+1 check for item(s) on current tile and remove one
- **+1/2** attempt on current tile (might try to remove all items)
- **+1/2** correct

+1 1/2 check required conditions in context of attempt to move/turn
(body of each check must refer to `pos` or `facingRight`)
- **+1** separate check for empty tile (e.g., not in ELSE)
- **+1/2** check `forwardMoveBlocked`

+1 change direction (set direction to some value relative to current direction)
- **+1/2** toggle value
- **+1/2** if and only if originally blocked

+1 1/2 move (set position to value(s) relative to current position)
- **+1/2** attempt 2 directions (change position, not value at position)
- **+1/2** move 1 tile in proper direction
- **+1/2** if and only if originally not blocked

Part C:	`clearHall`	**3 pts**

+1/2 declare and initialize counter (must have some extra context relevant to counting)

+1 loop until done
- **+1/2** call to `hallIsClear` in loop
- **+1/2** correct

+1 robot action (in context of a loop)
- **+1/2** call `move`
- **+1/2** correctly determine number of times `move` is called

+1/2 always return number of times `move` is called (no credit for returning 0 with no call to `move` in code)

Copyright © 2004 by College Entrance Examination Board. All rights reserved.
Visit apcentral.collegeboard.com (for AP professionals) and www.collegeboard.com/apstudents (for AP students and parents).

APPENDIX C — SAMPLE SEARCH AND SORT ALGORITHMS

Sequential Search

The Sequential Search Algorithm below finds the index of a value in an array of integers as follows:

1. Traverse `elements` until `target` is located, or the end of `elements` is reached.
2. If `target` is located, return the index of `target` in `elements`; Otherwise return `-1`.

```
/**
 * Finds the index of a value in an array of integers.
 *
 * @param elements an array containing the items to be searched.
 * @param target the item to be found in elements.
 * @return an index of target in elements if found; -1 otherwise.
 */
public static int sequentialSearch(int[] elements, int target)
{
  for (int j = 0; j < elements.length; j++)
  {
    if (elements[j] == target)
    {
      return j;
    }
  }

  return -1;
}
```


© 2014 The College Board. Visit the College Board on the Web: www.collegeboard.org.

Binary Search

The Binary Search Algorithm below finds the index of a value in an array of integers sorted in ascending order as follows:

1. Set `left` and `right` to the minimum and maximum indexes of `elements` respectively.
2. Loop until `target` is found, or `target` is determined not to be in `elements` by doing the following for each iteration:
   a. Set `middle` to the index of the middle item in `elements[left]` ... `elements[right]` inclusive.
   b. If `target` would have to be in `elements[left]` ... `elements[middle - 1]` inclusive, then set `right` to the maximum index for that range.
   c. Otherwise, if `target` would have to be in `elements[middle + 1]` ... `elements[right]` inclusive, then set `left` to the minimum index for that range.
   d. Otherwise, return `middle` because `target == elements[middle]`.
3. Return `-1` if `target` is not contained in `elements`.

```
/**
 * Find the index of a value in an array of integers sorted in ascending order.
 *
 * @param elements an array containing the items to be searched.
 *         Precondition: items in elements are sorted in ascending order.
 * @param target the item to be found in elements.
 * @return an index of target in elements if target found;
 *         -1 otherwise.
 */
public static int binarySearch(int[] elements, int target)
{
  int left = 0;
  int right = elements.length - 1;
  while (left <= right)
  {
    int middle = (left + right) / 2;
    if (target < elements[middle])
    {
      right = middle - 1;
    }
    else if (target > elements[middle])
    {
      left = middle + 1;
    }
    else
    {
      return middle;
    }
  }
  return -1;
}
```


© 2014 The College Board. Visit the College Board on the Web: www.collegeboard.org.

Selection Sort

The Selection Sort Algorithm below sorts an array of integers into ascending order as follows:

1. Loop from `j = 0` to `j = elements.length−2`, inclusive, completing `elements.length−1` passes.
2. In each pass, swap the item at index `j` with the minimum item in the rest of the array (`elements[j+1]` through `elements[elements.length−1]`).

At the end of each pass, items in `elements[0]` through `elements[j]` are in ascending order and each item in this sorted portion is at its final position in the array

```
/**
 * Sort an array of integers into ascending order.
 *
 * @param elements an array containing the items to be sorted.
 *
 * Postcondition: elements contains its original items and items in elements
 *                are sorted in ascending order.
 */
public static void selectionSort(int[] elements)
{
  for (int j = 0; j < elements.length − 1; j++)
  {
     int minIndex = j;
     for (int k = j + 1; k < elements.length; k++)
     {
       if (elements[k] < elements[minIndex])
       {
          minIndex = k;
       }
     }

     int temp = elements[j];
     elements[j] = elements[minIndex];
     elements[minIndex] = temp;
  }
}
```

© 2014 The College Board. Visit the College Board on the Web: www.collegeboard.org.

Insertion Sort

The Insertion Sort Algorithm below sorts an array of integers into ascending order as follows:

1. Loop from `j = 1` to `j = elements.length−1` inclusive, completing `elements.length−1` passes.
2. In each pass, move the item at index `j` to its proper position in `elements[0]` to `elements[j]`:
   a. Copy item at index `j` to `temp`, creating a "vacant" element at index `j` (denoted by `possibleIndex`).
   b. Loop until the proper position to maintain ascending order is found for `temp`.
   c. In each inner loop iteration, move the "vacant" element one position lower in the array.
3. Copy `temp` into the identified correct position (at `possibleIndex`).

At the end of each pass, items at `elements[0]` through `elements[j]` are in ascending order.

```
/**
 * Sort an array of integers into ascending order.
 *
 * @param elements an array containing the items to be sorted.
 *
 * Postcondition: elements contains its original items and items in elements
 *                are sorted in ascending order.
 */
public static void insertionSort(int[] elements)
{
  for (int j = 1; j < elements.length; j++)
  {
    int temp = elements[j];
    int possibleIndex = j;
    while (possibleIndex > 0 && temp < elements[possibleIndex - 1])
    {
      elements[possibleIndex] = elements[possibleIndex - 1];
      possibleIndex--;
    }

    elements[possibleIndex] = temp;
  }
}
```


© 2014 The College Board. Visit the College Board on the Web: www.collegeboard.org.

Merge Sort

The Merge Sort Algorithm below sorts an array of integers into ascending order as follows:

`mergeSort`

This top-level method creates the necessary temporary array and calls the `mergeSortHelper` recursive helper method.

`mergeSortHelper`

This recursive helper method uses the Merge Sort Algorithm to sort `elements[from]` ... `elements[to]` inclusive into ascending order:

1. If there is more than one item in this range,
   a. divide the items into two adjacent parts, and
   b. call `mergeSortHelper` to recursively sort each part, and
   c. call the `merge` helper method to merge the two parts into sorted order.
2. Otherwise, exit because these items are sorted.

`merge`

This helper method merges two adjacent array parts, each of which has been sorted into ascending order, into one array part that is sorted into ascending order:

1. As long as both array parts have at least one item that hasn't been copied, compare the first un-copied item in each part and copy the minimal item to the next position in `temp`.
2. Copy any remaining items of the first part to `temp`.
3. Copy any remaining items of the second part to `temp`.
4. Copy the items from `temp[from]` ... `temp[to]` inclusive to the respective locations in `elements`.

```
/**
 * Sort an array of integers into ascending order.
 *
 * @param elements an array containing the items to be sorted.
 *
 * Postcondition: elements contains its original items and items in elements
 *                are sorted in ascending order.
 */
public static void mergeSort(int[] elements)
{
  int n = elements.length;
  int[] temp = new int[n];
  mergeSortHelper(elements, 0, n - 1, temp);
}
```

© 2014 The College Board. Visit the College Board on the Web: www.collegeboard.org.

```
/**
 * Sorts elements[from] ... elements[to] inclusive into ascending order.
 *
 * @param elements an array containing the items to be sorted.
 * @param from the beginning index of the items in elements to be sorted.
 * @param to the ending index of the items in elements to be sorted.
 * @param temp a temporary array to use during the merge process.
 *
 * Precondition:
 *    (elements.length == 0 or
 *     0 <= from <= to <= elements.length) and
 *    elements.length == temp.length
 * Postcondition: elements contains its original items and the items in elements
 *               [from] ... <= elements[to] are sorted in ascending order.
 */
private static void mergeSortHelper(int[] elements,
                                    int from, int to, int[] temp)
{
  if (from < to)
  {
    int middle = (from + to) / 2;
    mergeSortHelper(elements, from, middle, temp);
    mergeSortHelper(elements, middle + 1, to, temp);
    merge(elements, from, middle, to, temp);
  }
}
```


© 2014 The College Board. Visit the College Board on the Web: www.collegeboard.org.

```
/**
 * Merges two adjacent array parts, each of which has been sorted into ascending
 * order, into one array part that is sorted into ascending order.
 *
 * @param elements an array containing the parts to be merged.
 * @param from the beginning index in elements of the first part.
 * @param mid the ending index in elements of the first part.
 *        mid+1 is the beginning index in elements of the second part.
 * @param to the ending index in elements of the second part.
 * @param temp a temporary array to use during the merge process.
 *
 * Precondition: 0 <= from <= mid <= to <= elements.length and
 *   elements[from] ... <= elements[mid] are sorted in ascending order and
 *   elements[mid + 1] ... <= elements[to] are sorted in ascending order and
 *   elements.length == temp.length
 * Postcondition: elements contains its original items and
 *   elements[from] ... <= elements[to] are sorted in ascending order and
 *   elements[0] ... elements[from - 1] are in original order and
 *   elements[to + 1] ... elements[elements.length - 1] are in original order.
 */
private static void merge(int[] elements,
                          int from, int mid, int to, int[] temp)
{
  int i = from;
  int j = mid + 1;
  int k = from;

  while (i <= mid && j <= to)
  {
    if (elements[i] < elements[j])
    {
      temp[k] = elements[i];
      i++;
    }
    else
    {
      temp[k] = elements[j];
      j++;
    }
    k++;
  }
```

© 2014 The College Board. Visit the College Board on the Web: www.collegeboard.org.

```
    while (i <= mid)
    {
      temp[k] = elements[i];
      i++;
      k++;
    }

    while (j <= to)
    {
      temp[k] = elements[j];
      j++;
      k++;
    }

    for (k = from; k <= to; k++)
    {
      elements[k] = temp[k];
    }
}
```


© 2014 The College Board. Visit the College Board on the Web: www.collegeboard.org.