Sorting Practice

The following method is a correct implementation of the insertion sort algorithm. The method correctly sorts the elements of numList so that they appear in order from least to greatest.

```
public static void insertionSort(ArrayList<Integer> numList)
{
  for (int j = 1; j < numList.size(); j++)
  {
    int temp = numList.get(j);
    int k = j;
    while (k > 0 && temp < numList.get(k - 1))
    {
      numList.set(k, numList.get(k - 1));
      k--;
    }
    numList.set(k, temp);
    /* end of outer loop */
  }
}
```

Assume that insertionSort has been called with an ArrayList parameter that has been initialized with the following Integer objects.

[60, 30, 10, 20, 50, 40]

1. What will the contents of numList be after three passes of the outer loop (i.e., when j == 3 at the point indicated by /* end of outer loop */)?

Sorting Practice

(A) [10, 20, 30, 50, 60, 40]

(B) [10, 20, 30, 60, 50, 40] ✓

(C) [10, 30, 60, 20, 50, 40]

(D) [20, 10, 30, 60, 50, 40]

Answer B

Correct. In each iteration of the outer loop, the method moves the element at index j to its proper position in the part of the list from index 0 to index j. After the first iteration of the outer loop, 30 is moved to index 0 so that the contents of numList are [30, 60, 10, 20, 50, 40]. After the second iteration of the outer loop, 10 is moved to index 0 so that the contents of numList are [10, 30, 60, 20, 50, 40]. After the third iteration of the outer loop, 20 is moved to index 1 so that the contents of numList are [10, 20, 30, 60, 50, 40].

Sorting Practice

Consider the following method. This method correctly sorts the elements of array data into increasing order.

```
public static void sort(int[] data)
{
  for (int j = 0; j < data.length - 1; j++)
  {
    int m = j;
    for (int k = j + 1; k < data.length; k++)
    {
      if (data[k] < data[m])    /* Compare values */
      {
        m = k;
      }
    }
    int temp = data[m];    /* Assign to temp */
    data[m] = data[j];
    data[j] = temp;
    /* End of outer loop */
  }
}
```

2. Assume that sort is called with the array {6, 3, 2, 5, 4, 1}. Which of the following represents the contents of data after three passes of the outer loop (i.e., when j == 2 at the point indicated by /* End of outer loop */)?

(A) {1, 2, 3, 4, 5, 6}

(B) {1, 2, 3, 5, 4, 6} ✓

(C) {1, 2, 3, 6, 5, 4}

(D) {1, 3, 2, 5, 4, 6}

Sorting Practice

Answer B

Correct. The method implements a selection sort algorithm. It looks for the smallest value in the array ranging from index j to the end of the array and swaps the smallest value with the value at position j. In the first pass, 1 is the smallest element from position j = 0 to the end of the array, so it is swapped with 6, resulting in {1, 3, 2, 5, 4, 6}. In the second pass, 2 is the smallest element from position j = 1 to the end of the array, so it is swapped with 3, resulting in {1, 2, 3, 5, 4, 6}. In the third pass, 3 is the smallest element from position j = 2 to the end of the array, so it is swapped with itself, resulting in {1, 2, 3, 5, 4, 6}.

3. Assume that sort is called with the array {1, 2, 3, 4, 5, 6}. How many times will the expression indicated by /* Compare values */ and the statement indicated by /* Assign to temp */ execute?

Sorting Practice

(A)

/* Compare values */	/* Assign to temp */
15 times	5 times

(B)

/* Compare values */	/* Assign to temp */
15 times	6 times

(C)

/* Compare values */	/* Assign to temp */
21 times	5 times

(D)

/* Compare values */	/* Assign to temp */
21 times	6 times

Answer A

Correct. The outer loop iterates one fewer time than the length of the array. The inner loop is dependent on the outer loop, starting at the value of `j + 1` and iterating to the end of the array. On the first pass through the outer loop, the inner loop will iterate five times, meaning the `/* Compare values */`

Sorting Practice

expression will execute 5 times. Each subsequent iteration will result in one fewer iteration of the inner loop. Therefore, the /* Compare values */ expression is evaluated 5 + 4 + 3 + 2 + 1, or 15, times. The /* Assign to temp */ expression is evaluated for every iteration of the outer loop. Therefore, the assignment is made 5 times.

The following method is a correct implementation of the selection sort algorithm. The method sorts the elements of arr so that they are in order from least to greatest.

```
public static void selectionSort(int[] arr)
{
  for (int j = 0; j < arr.length - 1; j++)
  {
    int minIndex = j;
    for (int k = j + 1; k < arr.length; k++)
    {
      if (arr[k] < arr[minIndex])
      {
        minIndex = k;
      }
    }

    int temp = arr[j];
    arr[j] = arr[minIndex];
    arr[minIndex] = temp;
    /* end of outer loop */
  }
}
```

Assume that selectionSort has been called with an int[] argument that has been initialized with the following contents.

{40, 30, 50, 60, 10, 20}

4. What will the contents of arr be after three iterations of the outer loop (i.e., when j == 2 at the point indicated by /* end of outer loop */)?

Sorting Practice

(A) {10, 20, 30, 40, 60, 50}

(B) {10, 20, 30, 60, 40, 50} ✓

(C) {10, 20, 50, 60, 40, 30}

(D) {10, 30, 50, 60, 40, 20}

Answer B

Correct. In each iteration of the outer loop, the method swaps the element at index `j` with the minimum element in the rest of the array (from `arr[j + 1]` to the end of the array). After the first iteration of the outer loop, `40` is swapped with `10` so that the contents of `arr` are {10, 30, 50, 60, 40, 20}. After the second iteration of the outer loop, `20` is swapped with `30` so that the contents of `arr` are {10, 20, 50, 60, 40, 30}. After the third iteration of the outer loop, `50` is swapped with `30` so that the contents of `arr` are {10, 20, 30, 60, 40, 50}.

Sorting Practice

Consider the following correct implementation of the selection sort algorithm.

```
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;
      }
    }
    if (j != minIndex)
    {
      int temp = elements[j];
      elements[j] = elements[minIndex];
      elements[minIndex] = temp;        // Line 17
    }
  }
}
```

The following declaration and method call appear in a method in the same class as selectionSort.

```
int[] arr = {9, 8, 7, 6, 5};
selectionSort(arr);
```

5. How many times is the statement in line 17 of the method executed as a result of the call selectionSort(arr)?

(A) 2 ✓

(B) 3

(C) 4

(D) 5

Sorting Practice

Answer A

Correct. The statement in line 17 executes each time a value is swapped into the correct position in the array. For the given array, the values 9 and 5 are swapped. Then the values 8 and 6 are swapped. Then, since 7, 8, and 9 are already in the correct position, no additional swaps occur.

The following method is intended to sort an array of integers into ascending order.

```
public static void sort(int[] arr)
{
  for (int j = 0; j < arr.length - 1; j++)
  {
    int minIndex = j;
    for (int k = j + 1; k < arr.length - 1; k++)
    {
      if (arr[k] < arr[minIndex])
      {
        minIndex = k;
      }
    }
    int temp = arr[minIndex];
    arr[minIndex] = arr[j];
    arr[j] = temp;
  }
}
```

6. This method works as intended for some, but not all, int arrays. Which of the following arrays can be used to show that the method does not work as intended?

Sorting Practice

(A) {1, 2, 3, 4, 5}

(B) {1, 2, 3, 5, 4} ✓

(C) {1, 2, 4, 3, 5}

(D) {1, 3, 2, 4, 5}

Answer B

Correct. The last element in the array is never processed by the method since the inner loop ends at arr.length - 1; thus, it will not be sorted when completed.

Sorting Practice

The following method is a correct implementation of the insertion sort algorithm. The method correctly sorts the elements of arr so that they appear in order from least to greatest.

```
public static void insertionSort(int[] arr)
{
  for (int j = 1; j < arr.length; j++)
  {
    int temp = arr[j];
    int possibleIndex = j;
    while (possibleIndex > 0 && temp < arr[possibleIndex - 1])
    {
      arr[possibleIndex] = arr[possibleIndex - 1];
      possibleIndex--;                    // Line 10
    }
    arr[possibleIndex] = temp;
  }
}
```

The following code segment appears in a method in the same class as insertionSort.

```
int[] numbers = {4, 12, 4, 7, 19, 6};
insertionSort(numbers);
```

7. How many times is the statement in line 10 of the method executed as a result of the call to insertionSort?

(A) 2

(B) 3

(C) 4

(D) 5 ✓

Answer D

Correct. The statement in line 10 is executed each time an element is moved in the array. In the second iteration of the for loop, the 12 located at index 1 is moved to index 2, and 4 is inserted at index 1. The array then contains {4, 4, 12, 7, 19, 6} and line 10 has executed 1 time. In the third iteration of the for loop, the 12 located at index 2 is moved to index 3, and 7 is inserted at index 2. The array then

Sorting Practice

contains {4, 4, 7, 12, 19, 6} and line 10 has executed a total of 2 times. In the fifth iteration of the for loop, the 19 located at index 4 is moved to index 5, the 12 located at index 3 is moved to index 4, the 7 located at index 2 is moved to index 3, and 6 is inserted at index 2. The array then contains {4, 4, 6, 7, 12, 19} and line 10 has executed 3 additional times, for a total of 5 times.

The following method is a correct implementation of the selection sort algorithm. The method correctly sorts the elements of data so that they appear in order from least to greatest.

```
public static void selectionSort(int[] data)
{
  for (int j = 0; j < data.length - 1; j++)
  {
    int minIndex = j;

    for (int k = j + 1; k < data.length; k++)
    {
      if (data[k] < data[minIndex])
      {
        minIndex = k;     // Line 11
      }
    }

    if (j != minIndex)
    {
      int temp = data[j];
      data[j] = data[minIndex];
      data[minIndex] = temp;
    }
  }
}
```

The following code segment appears in a method in the same class as selectionSort.

```
int[] vals = {5, 10, 2, 1, 12};
selectionSort(vals);
```

Sorting Practice

8. How many times is the statement in line 11 of the method executed as a result of the call to selectionSort?

(A) 1
(B) 2
(C) 3
(D) 4 ✓

Answer D

Correct. The statement in line 11 is executed when a new smallest element is found in the array. In the first iteration of the outer for loop, 5 (located at index 0) is currently the smallest element. While traversing through the rest of the array using the inner for loop, the code first sets 2 (located at index 2) to the new smallest element and then sets 1 (located at index 3) to the new smallest element. When the inner for loop terminates, 5 and 1 are swapped. The array now contains {1, 10, 2, 5, 12} and line 11 has executed 2 times. In the second iteration of the outer for loop, 10 (located at index 1) is the smallest element. While traversing through the rest of the array using the inner for loop, the code sets 2 (located at index 2) to the new smallest element. When the inner for loop terminates, 10 and 2 are swapped. The array now contains {1, 2, 10, 5, 12} and line 11 has executed 1 additional time, for a total of 3 times. In the third iteration of the outer for loop, 10 (located at index 2) is currently the smallest element. While traversing through the rest of the array using the inner for loop, the code sets 5 (located at index 3) to the smallest element. When the inner for loop terminates, 10 and 5 are swapped. The array now contains {1, 2, 5, 10, 12} and line 11 has executed 1 additional time, for a total of 4 times.

Sorting Practice

The following method is a correct implementation of the selection sort algorithm. The method correctly sorts the elements of `arr` so that they appear in order from least to greatest.

```
public static void selectionSort(int[] arr)
{
  for (int j = 0; j < arr.length - 1; j++)
  {
    int minIndex = j;
    for (int k = j + 1; k < arr.length; k++)
    {
      if (arr[k] < arr[minIndex])
      {
        minIndex = k;
      }
    }
    int temp = arr[j];
    arr[j] = arr[minIndex];
    arr[minIndex] = temp;
    /* end of outer loop */
  }
}
```

9. Which of the following could be the contents of `arr` after two passes of the outer loop (i.e., when `j == 1` at the point indicated by `/* end of outer loop */`)?

(A) {10, 20, 50, 40, 30} ✓

(B) {10, 30, 50, 40, 20}

(C) {20, 10, 50, 40, 30}

(D) {20, 30, 50, 40, 10}

Sorting Practice

Answer A

Correct. After two passes in selection sort, the smallest two elements must be in the first and second positions in the sorted array.

Sorting Practice

10. Consider the static method selectSort shown below. Method selectSort is intended to sort an array into increasing order; however, it does not always work as intended.

```
// precondition:  numbers.length > 0

// postcondition: numbers is sorted in increasing order

public static void selectSort(int[] numbers)

{

  int temp;

Line 1:   for (int j = 0; j < numbers.length - 1; j++)

   {

Line 2:    int pos = 0;

Line 3:    for (int k = j + 1; k < numbers.length; k++)

                {

Line 4:     if (numbers[k] < numbers[pos])

                    {

Line 5:      pos = k;

                     }

                }

   temp = numbers[j];

   numbers[j] = numbers[pos];

   numbers[pos] = temp;

  }

}
```

Which of the following changes should be made so that selectSort will work as intended?

Sorting Practice

(A) Line 1 should be changed to
```
for (int j = 0; j < numbers.length - 2; j++)
```

(B) Line 2 should be changed to ✓
```
int pos = j;
```

(C) Line 3 should be changed to
```
for (int k = 0; k < numbers.length; k++)
```

(D) Line 4 should be changed to
```
if (numbers[k] > numbers[pos])
```

Sorting Practice

11. The following incomplete method is intended to sort its array parameter arr in increasing order.

```
// postcondition: arr is sorted in increasing order
public static void sortArray(int[] arr)
{
  int j, k;

  for (j = arr.length - 1; j > 0; j--)
  {
    int pos = j;

    for ( /* missing code */ )
    {
      if (arr[k] > arr[pos])
      {
        pos = k;
      }
    }
    swap(arr, j, pos);
  }
}
```

Assume that swap(arr, j, pos) exchanges the values of arr[j] and arr[pos]. Which of the following could be used to replace /* *missing code* */ so that executing the code segment sorts the values in array arr?

Sorting Practice

(A) k = j - 1; k > 0; k--

(B) k = j - 1; k >= 0; k-- ✓

(C) k = 1; k < arr.length; k++

(D) k = 0; k <= arr.length; k++

Consider the following correct implementation of the selection sort algorithm.

```
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;
      }
    }
    if (j != minIndex)
    {
      int temp = elements[j];
      elements[j] = elements[minIndex];
      elements[minIndex] = temp;   // line 17
    }
  }
}
```

The following declaration and method call appear in a method in the same class as selectionSort.

```
int[] arr = {30, 40, 10, 50, 20};
selectionSort(arr);
```

12. How many times is the statement elements[minIndex] = temp; in line 17 of the method executed as a result of the call to selectionSort?

Sorting Practice

(A) 2

(B) 3 ✓

(C) 4

(D) 5

Answer B

Correct. The statement in line 17 executes each time a value is swapped into the correct position in the array. Three swaps would occur. For the given array, the values 30 and 10 are swapped. Then the values 40 and 20 are swapped. Then, since 30 is already in the correct position, no swap occurs. Lastly, 50 and 40 are swapped.

Sorting Practice

13. Consider the following `mergeSortHelper` method, which is part of an algorithm to recursively sort an array of integers.

```
/** Precondition: (arr.length == 0 or 0 <= from <= to <= arr.length)
 * arr.length == temp.length
 */
public static void mergeSortHelper(int[] arr,
   int from, int to, int[] temp)
{
     if (from < to)
     {
         int middle = (from + to) / 2;
         mergeSortHelper(arr, from, middle, temp);
         mergeSortHelper(arr, middle + 1, to, temp);
         merge(arr, from, middle, to, temp);
     }
}
```

The `merge` method is used to merge two halves of an array (`arr[from]` through `arr[middle]`, inclusive, and `arr[middle + 1]` through `arr[to]`, inclusive) when each half has already been sorted into ascending order. For example, consider the array `arr1`, which contains the values `{1, 3, 5, 7, 2, 4, 6, 8}`. The lower half of `arr1` is sorted in ascending order (elements `arr1[0]` through `arr1[3]`, or `{1, 3, 5, 7}`), as is the upper half of `arr1` (elements `arr1[4]` through `arr1[7]`, or `{2, 4, 6, 8}`). The array will contain the values `{1, 2, 3, 4, 5, 6, 7, 8}` after the method call `merge(arr1, 0, 3, 7, temp)`. The array `temp` is a temporary array declared in the calling program.

Consider the following code segment, which appears in a method in the same class as `mergeSortHelper` and `merge`.

```
int[] vals = {80, 50, 30, 20, 60, 70};
int[] temp = new int[vals.length];
mergeSortHelper(vals, 0, vals.length - 1, temp);
```

Which of the following represents the arrays merged the last time the `merge` method is executed as a result of the code segment above?

(A) `{20, 30, 50}` and `{60, 70, 80}` are merged to form `{20, 30, 50, 60, 70, 80}`.

(B) `{20, 50, 70}` and `{30, 60, 80}` are merged to form `{20, 30, 50, 60, 70, 80}`.

(C) `{20, 50, 70}` and `{30, 60, 80}` are merged to form `{20, 50, 70, 30, 60, 80}`.

(D) `{30, 50, 80}` and `{20, 60, 70}` are merged to form `{20, 30, 50, 60, 70, 80}`. ✓

Answer D

Correct. The `merge` method appears after both recursive calls in the `mergeSortHelper` method, so it is not called until the condition `from < to` evaluates to `false` and there are no more

Sorting Practice

recursive calls to execute. This occurs when the two parts of the array to merge each contain one element. The original array is split into the arrays `{80}`, `{50}`, `{30}`, `{20}`, `{60}`, and `{70}`. These are then merged to form the arrays `{50, 80}`, `{30}`, `{20, 60}`, and `{70}`. These are then merged to form the arrays `{30, 50, 80}` and `{20, 60, 70}`. Lastly, these are merged to form `{20, 30, 50, 60, 70, 80}`.

Consider the following mergeSortHelper method, which is part of a correctly implemented algorithm to recursively sort an array of integers.

```
/** Preconditions: (arr.length == 0 or
 *                  0 <= from <= to <= arr.length)
 *                  arr.length == temp.length
 */
public static void mergeSortHelper(int[] arr, int from,
                                   int to, int[] temp)
{
  if (from < to)
  {
    int middle = (from + to) / 2;
    mergeSortHelper(arr, from, middle, temp);
    mergeSortHelper(arr, middle + 1, to, temp);
    merge(arr, from, middle, to, temp);
  }
}
```

The merge method (not shown) is used to merge two halves of an array (arr[from] through arr[middle], inclusive, and arr[middle + 1] through arr[to], inclusive) when each half has already been sorted into ascending order. For example, if arr1 contains {1, 3, 5, 7, 2, 4, 6, 8}, then arr1 will contain {1, 2, 3, 4, 5, 6, 7, 8} after the method call merge(arr1, 0, 3, 7, temp).

Consider the following code segment, which appears in a method in the same class as mergeSortHelper and merge.

```
int[] numbers = {40, 10, 20, 30};
int[] temp = new int[numbers.length];
mergeSortHelper(numbers, 0, numbers.length - 1, temp);
```

Sorting Practice

14. How many times will the `merge` method be called as a result of running the code segment?

(A) 1

(B) 2

(C) 3 ✓

(D) 4

Answer C

Correct. The `merge` method appears after both recursive calls in the `mergeSortHelper` method, so it is not called until the condition `from < to` evaluates to `false` and there are no more recursive calls to execute. This occurs when the array parts to be merged are single elements. The `merge` method is called to merge, or sort, the elements from indices `0` to `0` (40) and the elements from indices `1` to `1` (10), resulting in {10, 40, 20, 30}. The `merge` method is called again to merge the elements from indices `2` to `2` (20) and the elements from indices `3` to `3` (30), resulting in {10, 40, 20, 30}. The `merge` method is called a third time to merge the elements from indices `0` to `1` (10, 40) and the elements from indices `2` to `3` (20, 30), resulting in {10, 20, 30, 40}.