Answers and Explanations

Bullets mark each step in the process of arriving at the correct solution.

1. The answer is E.
   - Let's picture the contents of our `ArrayList` in a table:

0	1	2	3	4	5
Oakland	Chicago	Milwaukee	Seattle	Denver	Boston

   - After the remove at index 2, we have this (notice how the indices have changed):

0	1	2	3	4
Oakland	Chicago	Seattle	Denver	Boston

   - After adding Detroit, we have:

0	1	2	3	4	5
Oakland	Chicago	Seattle	Denver	Boston	Detroit

- After the remove at index 4, we have:

0	1	2	3	4
Oakland	Chicago	Seattle	Denver	Detroit

- And then we add Cleveland:

0	1	2	3	4	5
Oakland	Chicago	Seattle	Denver	Detroit	Cleveland

2. The answer is E.

- Let's make tables to represent what happens as each statement is executed.

```
subjects.add("French");
```

0
French

```
subjects.add("History");
```

0	1
French	History

```
subjects.set(1, "English");
```

0	1
French	English

```
subjects.add("Art");
```

0	1	2
French	English	Art

```
subjects.remove(1);
```

0	1
French	Art

```
subjects.set(2, "Math");
```

Oops! Can't do it. Trying will generate an `IndexOutOfBoundsException`.

3. The answer is D.

- Our array starts as:

5	3	8	1	6	4	2	7

- The 5 is bigger than everything to its left (which is nothing), so let it be. Our algorithm starts with the 3.
- Take out the 3 and store it in a variable: tempKey = 3. Now we have a gap.

5		8	1	6	4	2	7

- Is 3 < 5? Yes, so move the 5 to the right.

	5	8	1	6	4	2	7

- There's nothing more to check, so pop the 3 into the open space the 5 left behind.

3	5	8	1	6	4	2	7

- That's the order after the first pass through the loop (question 3).

4. The answer is A.

- We start where we left off at the end of question 3.

3	5	8	1	6	4	2	7

- Now for the second pass. The 3 and 5 are in order; tempKey = 8, leaving a gap.

3	5		1	6	4	2	7

- Since 5 < 8, we put the 8 back into its original spot.

3	5	8	1	6	4	2	7

- Now for the third pass: 3, 5, 8 are sorted; tempKey = 1, leaving a gap.

3	5	8		6	4	2	7

- 1 < 8, move 8 into the space 1 left behind. 1 < 5, move 5 over. 1 < 3, move 3 over, and put 1 in the first spot.

1	3	5	8	6	4	2	7

- Now for the fourth pass: 1, 3, 5, 8 are sorted; tempKey = 6, leaving a gap.

1	3	5	8		4	2	7

- 6 < 8, move the 8 into the space 6 left behind. 5 < 6, so we've found the spot for 6 and we pop it in.

1	3	5	6	8	4	2	7

- Giving us the answer to question 4.
- Note: One thing to look for in the answers when trying to recognize Insertion Sort—the end of the array doesn't change until it is ready to be sorted. So the 4 2 7 in our example are in the same order that they were in at the beginning.

5. The answer is B.

- Our array starts as:

5	3	8	1	6	4	2	7

- We scan the array to find the smallest element: 1.
- Now swap the 1 and the 5.

1	3	8	5	6	4	2	7

- That's the order after the first pass through the loop (question 5).

6. The answer is E.

- We start where we left off at the end of question 5.
- Now for the second pass. Leave the 1 alone and scan for the smallest remaining item: 2.

1	3	8	5	6	4	2	7

- The 3 is in the spot the 2 needs to go into, so swap the 3 and the 2.

1	2	8	5	6	4	3	7

- The third pass swaps the 8 and the 3.

1	2	3	5	6	4	8	7

- The fourth pass swaps the 5 and the 4.

1	2	3	4	6	5	8	7

- This gives us the answer to question 6.

7. The answer is C.

- After the initial for loop, the contents of the `ArrayList` are: [10, 11, 12, 13].
- Let's consider what the `for-each` loop is doing. For each `Integer` (called i) in integerList
  - add it to total (notice that total starts at 3).
  - (total % 2 == 1) is a way of determining if a number is odd. If total is odd, subtract 1 from it.
- Executing the `for-each` loop gives us:
  - total = 3 + 10 = 13 . . . odd → 12
  - total = 12 + 11 = 23 . . . odd → 22
  - total = 22 + 12 = 34 . . . even!
  - total = 34 + 13 = 47 . . . odd → 46 and our final answer.

8. The answer is C.

- Using a loop to remove items from an `ArrayList` can be very tricky because when you remove an item, all the items after the removed item change indices. That's why option I does not work. When item 3 "words" is removed, "starting" shifts over and becomes item 3, but the loop marches on to item 4 (now "with") and so "starting" is never considered and never removed.
- Working backward through a loop works very well when removing items from an `ArrayList` because only items after the removed item will change indices, and you will have already looked at those items. Option II works and is the simplest solution.
- Option III also works because it only increments the loop counter when an item is not removed. If an item is removed, the loop counter remains the same, so the item that gets shifted down ("starting" in the example above) is not skipped. Option III uses a `while` loop because it changes the loop index i, and it is bad programming style to mess with the loop index inside a `for` loop.

9. The answer is B.

- There are many small alterations possible when writing any algorithm. Do you go from 1 to length or from 0 to length -1, for example. This algorithm makes those kinds of alterations to the version given in the text. But in all cases, the general approach is:
  - The outer loop goes through the array. Everything to the left of the outer loop index (i in this case) is sorted.
  - Take the next element and put it into a temporary variable (key in this case).
  - Look at the element to the left of the element you are working on. If it is larger than the element to be sorted, move it over. Keep doing that until you find an element that is smaller than the element to be sorted (this is the condition we need).
  - Now you've found where our element needs to go. Exit inner loop and pop the element into place.
- So the condition we are looking for reflects *keep doing that until we find an element that is smaller than the element to be sorted,* or, phrasing it like a `while` loop ("as long as" instead of "until"), *keep going as long as the element we are looking at is larger than our key.* In code, we want to keep going as long as arr[j] > key, so that's our condition.

10. Determine if the numbers in an `ArrayList` are always increasing.

Algorithm:
Step 1: Look at each of the numbers in the list starting at the second element
Step 2: If the first number is greater than or equal to the second number, then return false
Step 3: Advance to the next element and continue until you reach the end of the list
Step 4: Return true

Pseudocode:

```
for (iterate through all the numbers in the original list starting at the second element)
{
   if (list[index]<= list[index - 1])
   {
     return false
   }
}
return true
```

Java code:

```
public static boolean isIncreasing(ArrayList<Integer> arr)
{
   for (int i = 1; i < arr.size(); i++)
   {
      if (arr.get(i) <= arr.get(i - 1))     // if current <= previous
      {
         return false;                      // list is not increasing
      }
   }
   return true;
}
```

11. Determine if the rate is increasing.

Algorithm:
Step 1: Iterate through the parameter array starting at the third element
Step 2: If (the third element minus the second element) is less than or equal to the (second element minus the first element), then return false
Step 3: Move onto the next element and go to Step 2 (and adjust the elements #s)
Step 4: If you get this far without returning, return true

Pseudocode:

```
for (iterate through the parameter array starting with the third element (index 2))
{
   if ((list[index] – list[index – 1]) <= (list[index – 1] – list[index – 2]))
   {
     return false
   }
}
return true
```

Java code:

```
public static boolean rateIsIncreasing(ArrayList<Double> stockPrices)
{
   for (int i = 2; i < stockPrices.size(); i++) // start with the 3rd element
   {
      if ((stockPrices.get(i) - stockPrices.get(i - 1)) <=
          (stockPrices.get(i - 1) - stockPrices.get(i - 2)))
      {
         return false;
      }
   }
   return true;                  // the rate is increasing
}
```

12. Count the empty lockers.

Algorithm:

Step 1: Create a variable called `total` and set it to zero
Step 2: Look at each of the lockers in the list
Step 3: If the locker is not in use, increment the `total`
Step 4: Continue until you reach the end of the list
Step 5: Return the `total`

Pseudocode:

```
set total = 0
for (iterate through all the lockers in the list)
{
   if (the locker is not in use)
   {
      total = total + 1
   }
}
return total
```

Java Code:

```
public static int countEmptyLockers(ArrayList <Locker> lockers)
{
   int total = 0;
   for  (Locker element : lockers)
   {
      if (!element.isInUse())
      {
         total++;
      }
   }
   return total;
}
```