4_F1

SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.

Assume that the classes listed in the Java Quick Reference 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.
In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods will not receive full credit.

This question involves the use of *check digits*, which can be used to help detect if an error has occurred when a number is entered or transmitted electronically. An algorithm for computing a check digit, based on the digits of a number, is provided in part (a).

The `CheckDigit` class is shown below. You will write two methods of the `CheckDigit` class.

```
public class CheckDigit
{
     /** Returns the check digit for num, as described in part (a).
       * Precondition: The number of digits in num is between one and six,
   inclusive.
       * num >= 0
       */
     public static int getCheck(int num)
     {
         /* to be implemented in part (a) */
     }

     /** Returns true if numWithCheckDigit is valid, or false otherwise, as
  described in part (b).
      * Precondition: The number of digits in numWithCheckDigit is between two
  and seven, inclusive.
       * numWithCheckDigit >= 0
      */
     public static boolean isValid(int numWithCheckDigit)
     {
         /* to be implemented in part (b) */
     }

     /** Returns the number of digits in num. */
     public static int getNumberOfDigits(int num)
     {
         /* implementation not shown */
     }

     /** Returns the nth digit of num.
       * Precondition: n >= 1 and n <= the number of digits in num
       */
     public static int getDigit(int num, int n)
     {
```

4_F1

```
        /* implementation not shown */
    }

    // There may be instance variables, constructors, and methods not shown.
}
```

4_F1

1. **(a)** Write the `getCheck` method, which computes the check digit for a number according to the following rules.

Multiply the first digit by 7, the second digit (if one exists) by 6, the third digit (if one exists) by 5, and so on. The length of the method's `int` parameter is at most six; therefore, the last digit of a six-digit number will be multiplied by 2.
Add the products calculated in the previous step.
Extract the check digit, which is the rightmost digit of the sum calculated in the previous step.

The following are examples of the check-digit calculation.

Example 1, where num has the value 283415

The sum to calculate is
$(2x7) + (8x6) + (3x5) + (4x4) + (1x3) + (5x2) = 14 + 48 + 15 + 16 + 3 + 10 = 106$.
The check digit is the rightmost digit of 106, or 6, and `getCheck` returns the integer value `6`.

Example 2, where num has the value 2183

The sum to calculate is $(2x7) + (1x6) + (8x5) + (3x4) = 14 + 6 + 40 + 12 = 72$.
The check digit is the rightmost digit of 72, or 2, and `getCheck` returns the integer value `2`.

Two helper methods, `getNumberOfDigits` and `getDigit`, have been provided.

`getNumberOfDigits` returns the number of digits in its `int` parameter.
`getDigit` returns the `n`th digit of its `int` parameter.

The following are examples of the use of `getNumberOfDigits` and `getDigit`.

Method Call	Return Value	Explanation
`getNumberOfDigits(283415)`	`6`	The number `283415` has 6 digits.
`getDigit(283415, 1)`	`2`	The first digit of `283415` is `2`.
`getDigit(283415, 5)`	`1`	The fifth digit of `283415` is `1`.

Complete the `getCheck` method below. You must use `getNumberOfDigits` and `getDigit` appropriately to receive full credit.

```
/** Returns the check digit for num, as described in part (a).
 * Precondition: The number of digits in num is between one and six,
inclusive.
 * num >= 0
 */
public static int getCheck(int num)
```

(b) Write the `isValid` method. The method returns `true` if its parameter `numWithCheckDigit`, which represents a number containing a check digit, is valid, and `false` otherwise. The check digit is always the rightmost digit of `numWithCheckDigit`.

4_F1

The following table shows some examples of the use of `isValid`.

Method Call	Return Value	Explanation
`getCheck(159)`	`2`	The check digit for `159` is `2`.
`isValid(1592)`	`true`	The number `1592` is a valid combination of a number (`159`) and its check digit (`2`).
`isValid(1593)`	`false`	The number `1593` is not a valid combination of a number (`159`) and its check digit (`3`) because `2` is the check digit for `159`.

Complete method `isValid` below. Assume that `getCheck` works as specified, regardless of what you wrote in part (a). You must use `getCheck` appropriately to receive full credit.

```
/** Returns true if numWithCheckDigit is valid, or false otherwise, as
described in part (b).
 * Precondition: The number of digits in numWithCheckDigit is between two
 and seven, inclusive.
 * numWithCheckDigit >= 0
 */
public static boolean isValid(int numWithCheckDigit)
```

Part A – getCheck method

Select a point value to view scoring criteria, solutions, and/or examples and to score the response. +1 indicates a point earned and -1 indicates a general or question-specific penalty.

+1 [Skill 3.A] Calls getNumberOfDigits and getDigit.

+1 [Skill 3.C] Calculates one partial sum = digit i * (8 - i)

+1 [Skill 3.C] Calculates all partial sums (*in the context of a loop, no bounds errors*).

+1 [Skill 3.C] Returns the last digit of the calculated sum as the check digit.

-1 (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)

-1 (x) Local variables used but none declared

Canonical Solution:

4_F1

```
public static int getCheck(int num)
{
   int sum = 0;
   for (int i = 1; i <= getNumberOfDigits(num); i++)
   {
      sum += (8 - i) * getDigit(num, i);
   }
   return sum % 10;
}
```

0	1	2	3	4

Total number of points earned (minus penalties) is equal to 4.

- ☐ **+1** Calls `getNumberOfDigits` and `getDigit`. **(Points earned)**
- ☐ **+1** Calculates one partial sum = digit `i * (8 - i)`. **(Points earned)**
- ☐ **+1** Calculates all partial sums (*in the context of a loop, no bounds errors*). **(Points earned)**
- ☐ **+1** Returns the last digit of the calculated sum as the check digit. **(Points earned)**
- ☐ **-1** (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check) **(General penalties)**
- ☐ **-1** (x) Local variables used but none declared **(General penalties)**

Canonical Solution:

```
public static int getCheck(int num)
{
   int sum = 0;
   for (int i = 1; i <= getNumberOfDigits(num); i++)
   {
      sum += (8 - i) * getDigit(num, i);
   }
   return sum % 10;
}
```

Part B – isValid method

Select a point value to view scoring criteria, solutions, and/or examples and to score the response. +1 indicates a point earned and -1 indicates a general or question-specific penalty.

+1 [Skill 3.C] Obtains check digit of `numWithCheckDigit`.

+1 [Skill 3.C] Obtains number remaining in `numWithCheckDigit` after check digit removed.

+1 [Skill 3.A] Calls `getCheck` on number without check digit.

+1 [Skill 3.C] Compares check digit of `numWithCheckDigit` and return value from `getCheck`

4_F1

+1 [Skill 3.C] Returns `true` or `false` depending on the result of the previous comparison.

-1 (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)

-1 (x) Local variables used but none declared

Canonical Solution:

```
public static boolean isValid(int numWithCheckDigit)
{
   int check = numWithCheckDigit % 10;
   int num = numWithCheckDigit / 10;
   int newCheck = getCheck(num);
   if (check == newCheck)
   {
      return true;
   }
   else
   {
      return false;
   }
}
```

0	1	2	3	4	5

Total number of points earned (minus penalties) is equal to 5.

- [ ] **+1** Obtains check digit of `numWithCheckDigit`. **(Points earned)**
- [ ] **+1** Obtains number remaining in `numWithCheckDigit` after check digit removed. **(Points earned)**
- [ ] **+1** Calls `getCheck` on number without check digit. **(Points earned)**
- [ ] **+1** Compares check digit of `numWithCheckDigit` and return value from `getCheck`. **(Points earned)**
- [ ] **+1** Returns `true` or `false` depending on the result of the previous comparison. **(Points earned)**
- [ ] **-1** (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check) **(General penalties)**
- [ ] **-1** (x) Local variables used but none declared **(General penalties)**

Canonical Solution:

4_F1

```
public static boolean isValid(int numWithCheckDigit)
{
   int check = numWithCheckDigit % 10;
   int num = numWithCheckDigit / 10;
   int newCheck = getCheck(num);
   if (check == newCheck)
   {
      return true;
   }
   else
   {
      return false;
   }
}
```

4_F1

2. This question involves reasoning about a simulation of a frog hopping in a straight line. The frog attempts to hop to a goal within a specified number of hops. The simulation is encapsulated in the following FrogSimulation class. You will write two of the methods in this class.

```
public class FrogSimulation
{
   /** Distance, in inches, from the starting position to the goal. */
   private int goalDistance;

   /** Maximum number of hops allowed to reach the goal. */
   private int maxHops;


   /** Constructs a FrogSimulation where dist is the distance, in inches, from the starting
    *  position to the goal, and numHops is the maximum number of hops allowed to reach the goal.
    *  Precondition: dist > 0; numHops > 0
    */
   public FrogSimulation(int dist, int numHops)
   {
      goalDistance = dist;
      maxHops = numHops;
   }


   /** Returns an integer representing the distance, in inches, to be moved when the frog hops.
    */
   private int hopDistance()
   {  /* implementation not shown */  }


   /** Simulates a frog attempting to reach the goal as described in part (a).
    *  Returns true if the frog successfully reached or passed the goal during the simulation;
    *          false otherwise.
    */
   public boolean simulate()
   {  /* to be implemented in part (a) */  }


   /** Runs num simulations and returns the proportion of simulations in which the frog
    *  successfully reached or passed the goal.
    *  Precondition: num > 0
    */
   public double runSimulations(int num)
   {  /* to be implemented in part (b) */  }
}
```

a. Write the simulate method, which simulates the frog attempting to hop in a straight line to a goal from the frog's starting position of 0 within a maximum number of hops. The method returns true if the frog successfully reached the goal within the maximum number of hops; otherwise, the method returns false.

   The FrogSimulation class provides a method called hopDistance that returns an integer representing the distance (positive or negative) to be moved when the frog hops. A positive distance represents a move toward the goal. A negative distance represents a move away from the goal. The returned distance may vary from call to call. Each time the frog hops, its position is adjusted by the value returned by a call to the hopDistance method.

   The frog hops until one of the following conditions becomes true:

   • The frog has reached or passed the goal.

   • The frog has reached a negative position.

   • The frog has taken the maximum number of hops without reaching the goal.

4_F1

The following example shows a declaration of a FrogSimulation object for which the goal distance is 24 inches and the maximum number of hops is 5. The table shows some possible outcomes of calling the simulate method.

FrogSimulation sim = new FrogSimulation(24, 5);

	Values returned by `hopDistance()`	Final position of frog	Return value of `sim.simulate()`
Example 1	`5, 7, -2, 8, 6`	24	`true`
Example 2	`6, 7, 6, 6`	25	`true`
Example 3	`6, -6, 31`	31	`true`
Example 4	`4, 2, -8`	-2	`false`
Example 5	`5, 4, 2, 4, 3`	18	`false`

Class information for this question

```
public class FrogSimulation

private int goalDistance
private int maxHops

private int hopDistance()
public boolean simulate()
public double runSimulations(int num)
```

Complete method simulate below. You must use hopDistance appropriately to receive full credit.

/** Simulates a frog attempting to reach the goal as described in part (a). * Returns true if the frog successfully reached or passed the goal during the simulation; * false otherwise. */

public boolean simulate()

b. Write the runSimulations method, which performs a given number of simulations and returns the proportion of simulations in which the frog successfully reached or passed the goal. For example, if the parameter passed to runSimulations is 400, and 100 of the 400 simulate method calls returned true, then the runSimulations method should return 0.25.

4_F1

Complete method runSimulations below. Assume that simulate works as specified, regardless of what you wrote in part (a). You must use simulate appropriately to receive full credit.

```
/** Runs num simulations and returns the proportion of simulations in which the frog
 *  successfully reached or passed the goal.
 *  Precondition: num > 0
 */
public double runSimulations(int num)
```

Part A

Intent: Simulate the distance traveled by a hopping frog

1 point is earned for: Calls hopDistance and uses returned distance to adjust (or represent) the frog's position

1 point is earned for: Initializes and accumulates the frog's position at most maxHops times (must be in context of a loop)

1 point is earned for: Determines if a distance representing multiple hops is at least goalDistance

1 point is earned for: Determines if a distance representing multiple hops is less than starting position

1 point is earned for: Returns true if goal ever reached, false if goal never reached or position ever less than starting position

```
public boolean simulate()
{
    int position = 0;
    for (int count = 0; count < maxHops; count++)
        {
        position += hopDistance();
        if (position >= goalDistance)
            {
            return true;
            }
            else if (position < 0)
            {
            return false;
```

4_F1

```
            }
        }
    return false;
}
```

0	1	2	3	4	✓ 5

Student response earns 5 of the following 5 point(s)

Intent: Simulate the distance traveled by a hopping frog

1 point is earned for: Calls hopDistance and uses returned distance to adjust (or represent) the frog's position

1 point is earned for: Initializes and accumulates the frog's position at most maxHops times (must be in context of a loop)

1 point is earned for: Determines if a distance representing multiple hops is at least goalDistance

1 point is earned for: Determines if a distance representing multiple hops is less than starting position

1 point is earned for: Returns true if goal ever reached, false if goal never reached or position ever less than starting position

```
public boolean simulate()
{
    int position = 0;
    for (int count = 0; count < maxHops; count++)
        {
        position += hopDistance();
        if (position >= goalDistance)
            {
            return true;
            }
            else if (position < 0)
            {
```

4_F1

```
                return false;
                }
            }
        return false;
}
```

Part B

Intent: Determine the proportion of successful frog hopping simulations

1 point is earned for: Calls simulate the specified number of times (no bounds errors)

1 point is earned for: Initializes and accumulates a count of true results

1 point is earned for: Calculates proportion of successful simulations using double arithmetic

1 point is earned for: Returns calculated value

```
public double runSimulations(int num)
{
    int countSuccess = 0;
    for (int count = 0; count < num; count++)
    {
        if(simulate())
        {
        countSuccess++;
        }
    }
    return (double)countSuccess / num;
}
```

0	1	2	3	4 ✓

Student response earns 4 of the following 4 point(s)

4_F1

Intent: Determine the proportion of successful frog hopping simulations

1 point is earned for: Calls simulate the specified number of times (no bounds errors)

1 point is earned for: Initializes and accumulates a count of true results

1 point is earned for: Calculates proportion of successful simulations using double arithmetic

1 point is earned for: Returns calculated value

```
public double runSimulations(int num)
{
    int countSuccess = 0;
    for (int count = 0; count < num; count++)
    {
        if(simulate())
        {
        countSuccess++;
        }
    }
    return (double)countSuccess / num;
}
```

4_F1

3. SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.

- Assume that the classes listed in the Java Quick Reference 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.
- In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods will not receive full credit.

A mathematical sequence is an ordered list of numbers. This question involves a sequence called a *hailstone sequence*. If n is the value of a term in the sequence, then the following rules are used to find the next term, if one exists.

- If n is 1, the sequence terminates.
- If n is even, then the next term is $\frac{n}{2}$.
- If n is odd, then the next term is $3n + 1$.

For this question, assume that when the rules are applied, the sequence will eventually terminate with the term $n = 1$.

The following are examples of hailstone sequences.

Example 1: 5, 16, 8, 4, 2, 1

- The first term is 5, so the second term is $5 * 3 + 1 = 16$.
- The second term is 16, so the third term is $\frac{16}{2} = 8$.
- The third term is 8, so the fourth term is $\frac{8}{2} = 4$.
- The fourth term is 4, so the fifth term is $\frac{4}{2} = 2$.
- The fifth term is 2, so the sixth term is $\frac{2}{2} = 1$.
- Since the sixth term is 1, the sequence terminates.

Example 2: 8, 4, 2, 1

- The first term is 8, so the second term is $\frac{8}{2} = 4$.
- The second term is 4, so the third term is $\frac{4}{2} = 2$.
- The third term is 2, so the fourth term is $\frac{2}{2} = 1$.
- Since the fourth term is 1, the sequence terminates.

The `Hailstone` class, shown below, is used to represent a hailstone sequence. You will write three methods in the `Hailstone` class.

```
public class Hailstone
{
     /** Returns the length of a hailstone sequence that starts with n,
       * as described in part (a).
       * Precondition: n > 0
       */
     public static int hailstoneLength(int n)
```

4_F1

```
    { /* to be implemented in part (a) */ }
  /** Returns true if the hailstone sequence that starts with n is
considered long
      * and false otherwise, as described in part (b).
      * Precondition: n > 0
      */
    public static boolean isLongSeq(int n)
    { /* to be implemented in part (b) */ }
  /** Returns the proportion of the first n hailstone sequences that are
considered long,
      * as described in part (c).
      * Precondition: n > 0
      */
    public static double propLong(int n)
    { /* to be implemented in part (c) */ }
  // There may be instance variables, constructors, and methods not
shown.
}
```

(a) The length of a hailstone sequence is the number of terms it contains. For example, the hailstone sequence in example 1 (5, 16, 8, 4, 2, 1) has a length of 6 and the hailstone sequence in example 2 (8, 4, 2, 1) has a length of 4.

Write the method `hailstoneLength(int n)`, which returns the length of the hailstone sequence that starts with `n`.

```
/** Returns the length of a hailstone sequence that starts with n,
 * as described in part (a).
 * Precondition: n > 0
 */
public static int hailstoneLength(int n)
```

Class information for this question

```
public class Hailstone
public static int hailstoneLength(int n)
public static boolean isLongSeq(int n)
public static double propLong(int n)
```

(b) A hailstone sequence is considered long if its length is greater than its starting value. For example, the hailstone sequence in example 1 (5, 16, 8, 4, 2, 1) is considered long because its length (6) is greater than its starting value (5). The hailstone sequence in example 2 (8, 4, 2, 1) is not considered long because its length (4) is less than or equal to its starting value (8).

4_F1

Write the method `isLongSeq(int n)`, which returns `true` if the hailstone sequence starting with `n` is considered long and returns `false` otherwise. Assume that `hailstoneLength` works as intended, regardless of what you wrote in part (a). You must use `hailstoneLength` appropriately to receive full credit.

```
/** Returns true if the hailstone sequence that starts with n is
considered long
 * and false otherwise, as described in part (b).
 * Precondition: n > 0
 */
public static boolean isLongSeq(int n)
```

(c) The method `propLong(int n)` returns the proportion of long hailstone sequences with starting values between 1 and `n`, inclusive.

Consider the following table, which provides data about the hailstone sequences with starting values between 1 and 10, inclusive.

Starting Value	Terms in the Sequence	Length of the Sequence	Long?
1	1	1	No
2	2, 1	2	No
3	3, 10, 5, 16, 8, 4, 2, 1	8	Yes
4	4, 2, 1	3	No
5	5, 16, 8, 4, 2, 1	6	Yes
6	6, 3, 10, 5, 16, 8, 4, 2, 1	9	Yes
7	7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1	17	Yes
8	8, 4, 2, 1	4	No
9	9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1	20	Yes
10	10, 5, 16, 8, 4, 2, 1	7	No

The method call `Hailstone.propLong(10)` returns `0.5`, since 5 of the 10 hailstone sequences shown in the table are considered long.

Write the `propLong` method. Assume that `hailstoneLength` and `isLongSeq` work as intended, regardless of what you wrote in parts (a) and (b). You must use `isLongSeq` appropriately to receive full credit.

```
/** Returns the proportion of the first n hailstone sequences that are
considered long,
```

4_F1

```
 * as described in part (c).
 * Precondition: n > 0
 */
public static double propLong(int n)
```

Class information for this question

```
public class Hailstone
public static int hailstoneLength(int n)
public static boolean isLongSeq(int n)
public static double propLong(int n)
```

Part A – hailstoneLength method

Select a point value to view scoring criteria, solutions, and/or examples and to score the response. +1 indicates a point earned and -1 indicates a general or question-specific penalty.

+1 [Skill 3.C] Loops from given starting value n until the sequence terminates, using updated values for the current term

Responses still earn the point even if they...

· update n incorrectly.

+1 [Skill 3.C] Computes the next value

Responses still earn the point even if they...

· use a correct formula in an incorrect case.

+1 [Skill 3.C] Uses correct formula for next value depending on even/odd

-1 (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)

-1 (x) Local variables used but none declared

Canonical Solution:

4_F1

```
public static int hailstoneLength(int n)
{
   int count = 1;
   while (n > 1)
   {
      if (n % 2 == 0)
      {
         n = n / 2;
      }
      else
      {
         n = 3 * n + 1;
      }
      count++;
   }
   return count;
}
```

1) Line 5: { Line 6: if (n % 2 == 0) Line 7: { Line 8: n = n / 2; Line 9: } Line 10: else Line 11: { Line 12: n = 3 * n + 1; Line 13: } Line 14: count ++; Line 15: } Line 16: return count; Line 17: }" title="">

0	1	2	3 ✓

Total number of points earned (minus penalties) is equal to 3.

- ☐ **+1** Loops from given starting value n until the sequence terminates, using updated values for the current term **(Points earned)**
- ☐ **+1** Computes the next value **(Points earned)**
- ☐ **+1** Uses correct formula for next value depending on even/odd **(Points earned)**
- ☐ **-1** [penalty] (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check) **(General penalties)**
- ☐ **-1** [penalty] (x) Local variables used but none declared **(General penalties)**

Canonical Solution:

4_F1

```
public static int hailstoneLength(int n)
{
   int count = 1;
   while (n > 1)
   {
      if (n % 2 == 0)
      {
         n = n / 2;
      }
      else
      {
         n = 3 * n + 1;
      }
      count++;
   }
   return count;
}
```

1) Line 5: { Line 6: if (n % 2 == 0) Line 7: { Line 8: n = n / 2; Line 9: } Line 10: else Line 11: { Line 12: n = 3 * n + 1; Line 13: } Line 14: count ++; Line 15: } Line 16: return count; Line 17: }">

Part B – isLongSeq method

Select a point value to view scoring criteria, solutions, and/or examples and to score the response. +1 indicates a point earned and -1 indicates a general or question-specific penalty.

+1 [Skill 3.A] Calls hailstoneLength

+1 [Skill 3.C] Correctly compares length and starting value to determine return value

Responses still earn the point even if they...

· call hailstoneLength incorrectly.

-1 (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)

-1 (x) Local variables used but none declared

Canonical Solution:

```
public static boolean isLongSeq(int n)
{
   return hailstoneLength(n) > n;
}
```

n; Line 4: }">

0	1	2

Total number of points earned (minus penalties) is equal to 2.

4_F1

- [ ] **+1** Calls hailstoneLength **(Points earned)**
- [ ] **+1** Correctly compares length and starting value to determine return value **(Points earned)**
- [ ] **-1** [penalty] (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check) **(General penalties)**
- [ ] **-1** [penalty] (x) Local variables used but none declared **(General penalties)**

Canonical Solution:

```
public static boolean isLongSeq(int n)
{
   return hailstoneLength(n) > n;
}
```

n; Line 4: }">

Part C – propLong method

Select a point value to view scoring criteria, solutions, and/or examples and to score the response. +1 indicates a point earned and -1 indicates a general or question-specific penalty.

+1 [Skill 3.A] Calls isLongSeq in the context of a loop

+1 [Skill 3.C] Loops 1 to n (*no bounds errors*)

+1 [Skill 3.C] Calculates double proportion

Responses still earn the point even if they...

· use incorrect values for the count of long sequences or n.

+1 [Skill 3.B] Returns correctly calculated value

-1 (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)

-1 (x) Local variables used but none declared

Canonical Solution:

```
public static double propLong(int n)
{
   int count = 0;
   for (int i = 1; i <= n; i++)
   {
      if (isLongSeq(i))
      {
         count++;
      }
   }
   return (double) count / n;
}
```

4_F1

0	1	2	3	4 ✓

Total number of points earned (minus penalties) is equal to 4.

- ☐ **+1** Calls isLongSeq in the context of a loop **(Points earned)**
- ☐ **+1** Loops 1 to n (*no bounds errors*) **(Points earned)**
- ☐ **+1** Calculates double proportion **(Points earned)**
- ☐ **+1** Returns correctly calculated value **Points earned)**
- ☐ **-1** [penalty] (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check) **(General penalties)**
- ☐ **-1** [penalty] (x) Local variables used but none declared **(General penalties)**

Canonical Solution:

```
public static double propLong(int n)
{
   int count = 0;
   for (int i = 1; i <= n; i++)
   {
      if (isLongSeq(i))
      {
         count++;
      }
   }
   return (double) count / n;
}
```

4_F1

4. SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.

Assume that the classes listed in the Java Quick Reference 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.
In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods will not receive full credit.

This question involves the `StringManip` class, which is used to perform manipulation on strings.

The class provides the `removeSpaces` method, whose implementation is not shown. The method takes a string and returns a new string with spaces removed. For example, `removeSpaces("hi how are you")` returns `"hihowareyou"`. The `removeSpaces` method will be used in part (b).

```
public class StringManip
{
     /** Takes a string str and returns a new string
       * with all spaces removed.
       */
     public static String removeSpaces(String str)
     { /* implementation not shown */ }

     /** Takes a string str and returns a new string
       * with the characters reversed, as described in part (a).
       */
     public static String reverseString(String str)
     { /* to be implemented in part (a) */ }

     /** Determines whether str is a palindrome and prints a message
       * indicating the result, as described in part (b).
       * Precondition: str contains only lowercase letters and spaces.
       */
     public static void palindromeChecker(String str)
     { /* to be implemented in part (b) */ }
}
```

(a) Write method `reverseString`, which takes a string `str` and returns a new string with the characters in `str` in reverse order. For example, `reverseString("ABCDE")` should return `"EDCBA"`.

Complete the `reverseString` method below by assigning the reversed string to `result`.

```
/** Takes a string str and returns a new string
 * with the characters reversed.
 */
```

4_F1

```
public static String reverseString(String str)
{
     String result = "";
     return result;
}
```

For this question, let a *palindrome* be defined as a string that, when spaces are removed, reads the same forward and backward. For example, `"race car"` and `"taco cat"` are palindromes. You will write method `palindromeChecker`, which determines whether a string is a palindrome and prints a message indicating the result. Examples of the intended behavior of the method are shown in the following table.

Method Call	Printed Message
`palindromeChecker("taco cat")`	`taco cat is a palindrome`
`palindromeChecker("laid on no dial")`	`laid on no dial is a palindrome`
`palindromeChecker("level up")`	`level up is not a palindrome`

(b) Write method `palindromeChecker` below. Assume that `reverseString` works as specified, regardless of what you wrote in part (a). You must use `reverseString` and `removeSpaces` appropriately to receive full credit. Your implementation must conform to the examples in the table.

```
/** Determines whether str is a palindrome and prints a message
 * indicating the result, as described in part (b).
 * Precondition: str contains only lowercase letters and spaces.
 */
public static void palindromeChecker(String str)
```

Part A - reverseString method (4 points)

Points earned:

+1 [Skill 3.A] Examines an individual character in `str` using substring or another appropriate method

+1 [Skill 3.C] Iterates over each character in `str` without bounds error

+1 [Skill 3.C] Uses concatenation to build a new string from characters in `str`

+1 [Skill 3.C] Assigns correct reversed string to `result` to be returned

General Penalties:

-1 (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)

Canonical Solution:

4_F1

```
public String reverseString(String str)
{
   String result = "";
   for (int j = 0; j < str.length(); j++)
   {
      result = str.substring(j, j + 1) + result;
   }
   return result;
}
```

Alternate Canonical Solution:

```
public String reverseString(String str)
{
   String result = "";
   for (int j = str.length() - 1; j >= 0; j--)
   {
      result += str.substring(j, j + 1);
   }
   return result;
}
```

= 0; j--) Line 5: { Line 6: result += str.substring(j, j + 1); Line 7: } Line 8: return result; Line 9: }">

0	1	2	3	4

Total number of points earned (minus penalties) is equal to 4.

- ☐ +1 Examines an individual character in `str` using substring or another appropriate method **(Points Earned)**
- ☐ +1 Iterates over each character in `str` without bounds error **(Points Earned)**
- ☐ +1 Uses concatenation to build a new string from characters in `str` **(Points Earned)**
- ☐ +1 Assigns correct reversed string to `result` to be returned **(Points Earned)**
- ☐ -1 (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check) **(General Penalties)**

Canonical Solution:

```
public String reverseString(String str)
{
   String result = "";
   for (int j = 0; j < str.length(); j++)
   {
      result = str.substring(j, j + 1) + result;
   }
   return result;
}
```

4_F1

Alternate Canonical Solution:

```
public String reverseString(String str)
{
   String result = "";
   for (int j = str.length() - 1; j >= 0; j--)
   {
      result += str.substring(j, j + 1);
   }
   return result;
}
```

= 0; j--) Line 5: { Line 6: result += str.substring(j, j + 1); Line 7: } Line 8: return result; Line 9: }">

Part B - palindromeChecker method (5 points)

Points Earned:

+1 [Skill 3.A] Removes spaces from the string to be compared but retains spaces for the string used in printed message

Response earn this point if it…

· reverses the string before removing the spaces

+1 [Skill 3.A] Calls reverseString to obtain a reversed string

Response earn this point if it…

· reverses the string before removing the spaces

Response earns this point, but incurs general penalty x below if it...

· uses a local variable for the reversed string, but does not declare it.

+1 [Skill 3.A] Compares a string to its reverse using equals or another appropriate method

+1 [Skill 3.C] Uses a condition statement to print one message when str is a palindrome and print another message otherwise

+1 [Skill 3.C] Prints messages in specified format

General Penalties:

-1 (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)

-1 (x) Local variables used but none declared

-1 (z) Void method or constructor that returns a value

Canonical Solution:

4_F1

```
public void palindromeChecker(String str)
{
   String noSpaceStr = removeSpaces(str);
   if (noSpaceStr.equals(reverseString(noSpaceStr)))
   {
      System.out.println(str + " is a palindrome");
   }
   else
   {
      System.out.println(str +
                         " is not a palindrome");
   }
}
```

0	1	2	3	4	5 ✓

Total number of points earned (minus penalties) is equal to 5.

- ☐ **+1** Removes spaces from the string to be compared but retains spaces for the string used in printed message **(Points Earned)**
- ☐ **+1** Calls reverseString to obtain a reversed string **(Points Earned)**
- ☐ **+1** Compares a string to its reverse using equals or another appropriate method **(Points Earned)**
- ☐ **+1** Uses a condition statement to print one message when str is a palindrome and print another message otherwise **(Points Earned)**
- ☐ **+1** Prints messages in specified format **(Points Earned)**
- ☐ **-1** (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check) **(General Penalties)**
- ☐ **-1** (x) Local variables used but none declared **(General Penalties)**
- ☐ **-1** (z) Void method or constructor that returns a value **(General Penalties)**

Canonical Solution:

```
public void palindromeChecker(String str)
{
   String noSpaceStr = removeSpaces(str);
   if (noSpaceStr.equals(reverseString(noSpaceStr)))
   {
      System.out.println(str + " is a palindrome");
   }
   else
   {
      System.out.println(str +
                         " is not a palindrome");
   }
}
```

4_F1

5. SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.

- Assume that the classes listed in the Java Quick Reference 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.
- In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods will not receive full credit.

A manufacturer wants to keep track of the average of the ratings that have been submitted for an item using a running average. The algorithm for calculating a running average differs from the standard algorithm for calculating an average, as described in part (a).

A partial declaration of the `RunningAverage` class is shown below. You will write two methods of the `RunningAverage` class.

```
public class RunningAverage
{
     /** The number of ratings included in the running average. */
     private int count;

     /** The average of the ratings that have been entered. */
     private double average;

     // There are no other instance variables.

     /** Creates a RunningAverage object.
       * Postcondition: count is initialized to 0 and average is
       * initialized to 0.0.
       */
     public RunningAverage()
     { /* implementation not shown */ }

     /** Updates the running average to reflect the entry of a new
       * rating, as described in part (a).
       */
     public void updateAverage(double newVal)
     { /* to be implemented in part (a) */ }

     /** Processes num new ratings by considering them for inclusion
       * in the running average and updating the running average as
       * necessary. Returns an integer that represents the number of
       * invalid ratings, as described in part (b).
       * Precondition: num > 0
       */
```

4_F1

```
    public int processNewRatings(int num)
    { /* to be implemented in part (b) */ }

    /** Returns a single numeric rating. */
    public double getNewRating()
    { /* implementation not shown */ }
}
```

(a) Write the method `updateAverage`, which updates the `RunningAverage` object to include a new rating. To update a running average, add the new rating to a calculated total, which is the number of ratings times the current running average. Divide the new total by the incremented count to obtain the new running average.

For example, if there are 4 ratings with a current running average of 3.5, the calculated total is 4 times 3.5, or 14.0. When a fifth rating with a value of 6.0 is included, the new total becomes 20.0. The new running average is 20.0 divided by 5, or 4.0.

Complete method `updateAverage`.

```
/** Updates the running average to reflect the entry of a new
 * rating, as described in part (a).
 */
public void updateAverage(double newVal)
```

(b) Write the `processNewRatings` method, which considers `num` new ratings for inclusion in the running average. A helper method, `getNewRating`, which returns a single rating, has been provided for you.

The running average must only be updated with ratings that are greater than or equal to zero. Ratings that are less than 0 are considered invalid and are not included in the running average.

The `processNewRatings` method returns the number of invalid ratings. See the table below for three examples of how calls to `processNewRatings` should work.

4_F1

Statement	Ratings Generated	processNewRatings Return Value	Comments
`processNewRatings(2)`	`2.5, 4.5`	`0`	Both new ratings are included in the running average.
`processNewRatings(1)`	`-2.0`	`1`	No new ratings are included in the running average.
`processNewRatings(4)`	`0.0, -2.2, 3.5, -1.5`	`2`	Two new ratings (`0.0` and `3.5`) are included in the running average.

Complete method `processNewRatings`. Assume that `updateAverage` works as specified, regardless of what you wrote in part (a). You must use `getNewRating` and `updateAverage` appropriately to receive full credit.

```
/** Processes num new ratings by considering them for inclusion
 * in the running average and updating the running average as
 * necessary. Returns an integer that represents the number of
 * invalid ratings, as described in part (b).
 * Precondition: num > 0
 */
public int processNewRatings(int num)
```

Part A – updateAverage

Select a point value to view scoring criteria, solutions, and/or examples and to score the response. +1 indicates a point earned and −1 indicates a general or question-specific penalty.

+1 **[Skill 3.C]** Computes total and updates total and count

Responses can still earn the point even if they fail to update instance variables

+1 **[Skill 3.C]** Calculates correct average

Responses can still earn the point even if the calculated total and/or count is incorrect

4_F1

Responses will not earn the point if the calculation results in a loss of precision (i.e. integer division)

+1 **[Skill 3.B]** Updates instance variables (`count` and `average`)

Responses can still earn the point even if they calculate `count` or `average` incorrectly

−1 (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)

−1 (x) Local variables used but none declared

−1 (y) Destruction of persistent data (e.g., changing value referenced by parameter)

−1 (z) Void method or constructor that returns a value

Canonical Solution:

```
public void updateAverage(double newVal)
{
   double tot = average * count;
   tot += newVal;
   count++;
   average = tot / count;
}
```

0	1	2	3 ✓

Total number of points earned (minus penalties) is equal to 3.

- ☐ +1 Computes total and updates total and count **(Points earned)**
- ☐ +1 Calculates correct average **(Points earned)**
- ☐ +1 Updates instance variables (`count` and `average`) **(Points earned)**
- ☐ −1 [penalty] (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check) **(General penalties)**
- ☐ −1 [penalty] (x) Local variables used but none declared **(General penalties)**
- ☐ −1 [penalty] (y) Destruction of persistent data (e.g., changing value referenced by parameter) **(General penalties)**
- ☐ −1 [penalty] (z) Void method or constructor that returns a value **(General penalties)**

Canonical Solution:

4_F1

```
public void updateAverage(double newVal)
{
   double tot = average * count;
   tot += newVal;
   count++;
   average = tot / count;
}
```

Part B – processNewRatings method

Select a point value to view scoring criteria, solutions, and/or examples and to score the response. +1 indicates a point earned and −1 indicates a general or question-specific penalty.

+1 **[Skill 3.C]** Loops num times (*no bounds errors*)

+1 **[Skill 3.A]** Calls getNewRating

Responses can still earn the point even if they call getNewRating more than once per iteration

Responses will not earn the point if they

· include parameters

· call getNewRating on an object or class other than this

+1 **[Skill 3.C]** Compares a rating and 0

Responses can still earn the point even if the comparison with 0 is incorrect

+1 **[Skill 3.A]** Calls updateAverage

Responses will not earn the point if they

· include a parameter of the incorrect type

· call updateAverage on an object or class other than this

+1 **[Skill 3.C]** Computes number of invalid ratings and includes only appropriate ratings in the average (*algorithm*)

Responses will not earn the point if they

· include 0 in the incorrect case

· reverse the direction of the comparison

· call getNewRating more than once per iteration

· do not initialize counter to 0

· do not put the computation in the context of iteration

· include negative ratings in running average

4_F1

· return early

+1 **[Skill 3.B]** Returns counted number of omitted values

Responses can still earn the point even if the returned count is incorrect

−1 (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)

−1 (x) Local variables used but none declared

−1 (y) Destruction of persistent data (e.g., changing value referenced by parameter)

−1 (z) Void method or constructor that returns a value

Canonical Solution:

```
public int processNewRatings(int num)
{
   int invalid = 0;
   for (int x = 0; x < num; x++)
   {
      double v = getNewRating();
      if (v >= 0)
      {
         updateAverage(v);
      }
      else
      {
         invalid++;
      }
   }
   return invalid;
}
```

= 0) Line 8: { Line 9: updateAverage(v); Line 10: } Line 11: else Line 12: { Line 13: invalid++; Line 14:} Line 15: } Line 16: return invalid; Line 17: } end code">

0	1	2	3	4	5	6 ✓

Total number of points earned (minus penalties) is equal to 6.

- ☐ +1 Loops num times (*no bounds errors*) **(Points earned)**
- ☐ +1 Calls getNewRating **(Points earned)**
- ☐ +1 Compares a rating and 0 **(Points earned)**
- ☐ +1 Calls updateAverage **(Points earned)**
- ☐ +1 Computes number of invalid ratings and includes only appropriate ratings in the average (*algorithm*) **(Points earned)**

4_F1

- ☐ +1 Returns counted number of omitted values **(Points earned)**
- ☐ −1 [penalty] (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check) **(General penalties)**
- ☐ −1 [penalty] (x) Local variables used but none declared **(General penalties)**
- ☐ −1 [penalty] (y) Destruction of persistent data (e.g., changing value referenced by parameter) **(General penalties)**
- ☐ −1 [penalty] (z) Void method or constructor that returns a value **(General penalties)**

Canonical Solution:

```
public int processNewRatings(int num)
{
   int invalid = 0;
   for (int x = 0; x < num; x++)
   {
      double v = getNewRating();
      if (v >= 0)
      {
         updateAverage(v);
      }
      else
      {
         invalid++;
      }
   }
   return invalid;
}
```

= 0) Line 8: { Line 9: updateAverage(v); Line 10: } Line 11: else Line 12: { Line 13: invalid++; Line 14:} Line 15: } Line 16: return invalid; Line 17: } end code">

4_F1

6. SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.

- Assume that the classes listed in the Java Quick Reference 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.
- In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods will not receive full credit.

This question involves the `WordMatch` class, which stores a secret string and provides methods that compare other strings to the secret string. You will write two methods in the `WordMatch` class.

```
public class WordMatch
{
     /** The secret string. */
     private String secret;

     /** Constructs a WordMatch object with the given secret string
       * of lowercase letters.
       */
     public WordMatch(String word)
     {
         /* implementation not shown */
     }

     /** Returns a score for guess, as described in part (a).
       * Precondition: 0 < guess.length() <= secret.length()
       */
     public int scoreGuess(String guess)
     { /* to be implemented in part (a) */ }

     /** Returns the better of two guesses, as determined by scoreGuess
       * and the rules for a tie-breaker that are described in part (b).
       * Precondition: guess1 and guess2 contain all lowercase letters.
       * guess1 is not the same as guess2.
       */
     public String findBetterGuess(String guess1, String guess2)
     { /* to be implemented in part (b) */ }
}
```

(a) Write the `WordMatch` method `scoreGuess`. To determine the score to be returned, `scoreGuess` finds the number of times that `guess` occurs as a substring of `secret` and then multiplies that number by the square of the length of `guess`. Occurrences of `guess` may overlap within `secret`.

Assume that the length of `guess` is less than or equal to the length of `secret` and that `guess` is not an

4_F1

empty string.

The following examples show declarations of a `WordMatch` object. The tables show the outcomes of some possible calls to the `scoreGuess` method.

```
WordMatch game = new WordMatch("mississippi");
```

Value of `guess`	Number of Substring Occurrences	Score Calculation: (Number of Substring Occurrences) x (Square of the Length of `guess`)	Return Value of `game.scoreGuess(guess)`
`"i"`	4	`4 * 1 * 1 = 4`	`4`
`"iss"`	2	`2 * 3 * 3 = 18`	`18`
`"issipp"`	1	`1 * 6 * 6 = 36`	`36`
`"mississippi"`	1	`1 * 11 * 11 = 121`	`121`

```
WordMatch game = new WordMatch("aaaabb");
```

4_F1

Value of `guess`	Number of Substring Occurrences	Score Calculation: (Number of Substring Occurrences) x (Square of the Length of `guess`)	Return Value of `game.scoreGuess(guess)`
`"a"`	4	`4 * 1 * 1 = 4`	`4`
`"aa"`	3	`3 * 2 * 2 = 12`	`12`
`"aaa"`	2	`2 * 3 * 3 = 18`	`18`
`"aabb"`	1	`1 * 4 * 4 = 16`	`16`
`"c"`	0	`0 * 1 * 1 = 0`	`0`

Complete the `scoreGuess` method.

```
/** Returns a score for guess, as described in part (a).
 * Precondition: 0 < guess.length() <= secret.length()
 */
public int scoreGuess(String guess)
```

(b) Write the `WordMatch` method `findBetterGuess`, which returns the better guess of its two `String` parameters, `guess1` and `guess2`. If the `scoreGuess` method returns different values for `guess1` and `guess2`, then the guess with the higher score is returned. If the `scoreGuess` method returns the same value for `guess1` and `guess2`, then the alphabetically greater guess is returned.

The following example shows a declaration of a `WordMatch` object and the outcomes of some possible calls to the `scoreGuess` and `findBetterGuess` methods.

```
WordMatch game = new WordMatch("concatenation");
```

4_F1

Method Call	Return Value	Explanation
`game.scoreGuess("ten");`	`9`	`1 * 3 * 3`
`game.scoreGuess("nation");`	`36`	`1 * 6 * 6`
`game.findBetterGuess("ten", "nation");`	`"nation"`	Since `scoreGuess` returns `36` for `"nation"` and `9` for `"ten"`, the guess with the greater score, `"nation"`, is returned.
`game.scoreGuess("con");`	`9`	`1 * 3 * 3`
`game.scoreGuess("cat");`	`9`	`1 * 3 * 3`
`game.findBetterGuess("con", "cat");`	`"con"`	Since `scoreGuess` returns `9` for both `"con"` and `"cat"`, the alphabetically greater guess, `"con"`, is returned.

Complete method `findBetterGuess`.

Assume that `scoreGuess` works as specified, regardless of what you wrote in part (a). You must use `scoreGuess` appropriately to receive full credit.

```
/** Returns the better of two guesses, as determined by scoreGuess
 * and the rules for a tie-breaker that are described in part (b).
 * Precondition: guess1 and guess2 contain all lowercase letters.
 * guess1 is not the same as guess2.
 */
public String findBetterGuess(String guess1, String guess2)
```

Part A – scoreGuess

Select a point value to view scoring criteria, solutions, and/or examples and to score the response. +1 indicates a point earned and −1 indicates a general or question-specific penalty.

4_F1

+1 **[Skill 3.C]** Compares guess to a substring of secret

Responses can still earn the point even if they only call secret.indexOf(guess)

Responses will not earn the point if they use == instead of equals

+1 **[Skill 3.A]** Uses a substring of secret with correct length for comparison with guess

Responses can still earn the point even if they

· only call secret.indexOf(guess)

· use == instead of equals

+1 **[Skill 3.C]** Loops through all necessary substrings of secret (*no bounds errors*)

Responses will not earn the point if they skip overlapping occurrences

+1 **[Skill 3.C]** Counts number of identified occurrences of guess within secret (*in the context of a condition involving both secret and guess*)

Responses can still earn the point even if they

· initialize count incorrectly or not at all

· identify occurrences incorrectly

+1 **[Skill 3.C]** Calculates and returns correct final score (*algorithm*)

Responses will not earn the point if they

· initialize count incorrectly or not at all

· fail to use a loop

· fail to compare guess to multiple substrings of secret

· count the same matching substring more than once

· use a changed or incorrect guess length when computing the score

−1 (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)

−1 (x) Local variables used but none declared

−1 (y) Destruction of persistent data (e.g., changing value referenced by parameter)

Canonical Solution:

4_F1

```
public int scoreGuess(String guess)
{
   int count = 0;

   for (int i = 0;
        i <= secret.length() - guess.length(); i++)
   {
      if (secret.substring(i,
          i + guess.length()).equals(guess))
      {
         count++;
      }
   }

   return count * guess.length() * guess.length();
}
```

0	1	2	3	4	5

Total number of points earned (minus penalties) is equal to 5.

- +1 Compares guess to a substring of secret **(Points Earned)**
- +1 Uses a substring of secret with correct length for comparison with guess **(Points Earned)**
- +1 Loops through all necessary substrings of secret (*no bounds errors*) **(Points Earned)**
- +1 Counts number of identified occurrences of guess within secret (*in the context of a condition involving both secret and guess*) **(Points Earned)**
- +1 Calculates and returns correct final score (*algorithm*) **(Points Earned)**
- −1 [penalty] (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check) **(General Penalties)**
- −1 [penalty] (x) Local variables used but none declared **(General Penalties)**
- −1 [penalty] (y) Destruction of persistent data (e.g., changing value referenced by parameter) **(General Penalties)**

Canonical Solution:

4_F1

```
public int scoreGuess(String guess)
{
   int count = 0;

   for (int i = 0;
        i <= secret.length() - guess.length(); i++)
   {
      if (secret.substring(i,
          i + guess.length()).equals(guess))
      {
         count++;
      }
   }

   return count * guess.length() * guess.length();
}
```

Part B – findBetterGuess method

Select a point value to view scoring criteria, solutions, and/or examples and to score the response. +1 indicates a point earned and −1 indicates a general or question-specific penalty.

+1 **[Skill 3.A]** Calls scoreGuess to get scores for guess1 and guess2

Responses will not earn the point if they

· fail to include parameters in the method calls

· call the method on an object or class other than this

+1 **[Skill 3.C]** Compares the scores

Responses will not earn the point if they

· only compare using == or ! =

· fail to use the result of the comparison in a conditional statement

+1 **[Skill 3.C]** Determines which of guess1 and guess2 is alphabetically greater

Responses can still earn the point even if they reverse the comparison

Responses will not earn the point if they

· reimplement compareTo incorrectly

· use result of compareTo as if boolean

+1 **[Skill 3.B]** Returns the identified guess1 or guess2 (*algorithm*)

Responses can still earn the point even if they

· call scoreGuess incorrectly

4_F1

· compare strings incorrectly

Responses will not earn the point if they

· reverse a comparison

· omit either comparison

· fail to return a guess in some case

−1 (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)

−1 (x) Local variables used but none declared

−1 (y) Destruction of persistent data (e.g., changing value referenced by parameter)

Canonical Solution:

```
public String findBetterGuess(String guess1,
                              String guess2)
{
   if (scoreGuess(guess1) > scoreGuess(guess2))
   {
      return guess1;
   }
   if (scoreGuess(guess2) > scoreGuess(guess1))
   {
      return guess2;
   }
   if (guess1.compareTo(guess2) > 0)
   {
      return guess1;
   }
   return guess2;
}
```

scoreGuess(guess2)) Line 4: { Line 5: return guess1; Line 6: } Line 7: if (scoreGuess(guess2) > scoreGuess(guess1)) Line 8: { Line 9: return guess2; Line 10: } Line 11: if (guess1.compareTo(guess2) > 0) Line 12: { Line 13: return guess1; Line 14:} Line 15: return guess2; Line 16: } end code">

0	1	2	3	4 ✓

Total number of points earned (minus penalties) is equal to 4.

- ☐ +1 Calls scoreGuess to get scores for guess1 and guess2 **(Points Earned)**
- ☐ +1 Compares the scores **(Points Earned)**
- ☐ +1 Determines which of guess1 and guess2 is alphabetically greater **(Points Earned)**
- ☐ +1 Returns the identified guess1 or guess2 (*algorithm*) **(Points Earned)**

4_F1

- [ ] −1 [penalty] (w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check) **(General Penalties)**
- [ ] −1 [penalty] (x) Local variables used but none declared **(General Penalties)**
- [ ] −1 [penalty] (y) Destruction of persistent data (e.g., changing value referenced by parameter) **(General Penalties)**

Canonical Solution:

```
public String findBetterGuess(String guess1,
                              String guess2)
{
   if (scoreGuess(guess1) > scoreGuess(guess2))
   {
      return guess1;
   }
   if (scoreGuess(guess2) > scoreGuess(guess1))
   {
      return guess2;
   }
   if (guess1.compareTo(guess2) > 0)
   {
      return guess1;
   }
   return guess2;
}
```

scoreGuess(guess2)) Line 4: { Line 5: return guess1; Line 6: } Line 7: if (scoreGuess(guess2) > scoreGuess(guess1)) Line 8: { Line 9: return guess2; Line 10: } Line 11: if (guess1.compareTo(guess2) > 0) Line 12: { Line 13: return guess1; Line 14:} Line 15: return guess2; Line 16: } end code">