Applying the Scoring Criteria
Apply the question scoring criteria first, which always takes precedence. Penalty points can only be deducted in a part of the question that has earned credit via the question rubric. No part of a question (a, b, c) may have a negative point total. A given penalty can be assessed only once for a question, even if it occurs multiple times or in multiple parts of that question. A maximum of 3 penalty points may be assessed per question.

1-Point Penalty
v) Array/collection access confusion ([ ] get)
w) Extraneous code that causes side-effect (e.g., printing to output, incorrect precondition check)
x) Local variables used but none declared
y) Destruction of persistent data (e.g., changing value referenced by parameter)
z) Void method or constructor that returns a value

No Penalty
• Extraneous code with no side-effect (e.g., valid precondition check, no-op)
• Spelling/case discrepancies where there is no ambiguity *
• Local variable not declared provided other variables are declared in some part
• private or public qualifier on a local variable
• Missing public qualifier on class or constructor header
• Keyword used as an identifier
• Common mathematical symbols used for operators (× • ÷ ≤ ≥ <> ≠)
• [ ] vs. () vs. <>
• = instead of == and vice versa
• length/size confusion for array, String, List, or ArrayList; with or without ( )
• Extraneous [ ] when referencing entire array
• [i,j] instead of [i][j]
• Extraneous size in array declaration, e.g., int[size] nums = new int[size];
• Missing ; where structure clearly conveys intent
• Missing { } where indentation clearly conveys intent
• Missing ( ) on parameter-less method or constructor invocations
• Missing ( ) around if or while conditions

*Spelling and case discrepancies for identifiers fall under the “No Penalty” category only if the correction can be unambiguously inferred from context, for example, “ArayList” instead of “ArrayList”. As a counterexample, note that if the code declares "int G=99, g=0;", then uses "while (G < 10)" instead of "while (g < 10)", the context does not allow for the reader to assume the use of the lower case variable.
Question 1: Methods and Control Structures  

Canonical solution

(a)    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(); 
    } 

(b)    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; 
    }
(a)    scoreGuess

<table>
<thead>
<tr>
<th>Scoring Criteria</th>
<th>Decision Rules</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1    Compares guess to a substring of secret</td>
<td>Responses can still earn the point even if they only call secret.indexOf(guess)</td>
<td>1 point</td>
</tr>
<tr>
<td></td>
<td>Responses will not earn the point if they use == instead of equals</td>
<td></td>
</tr>
<tr>
<td>2    Uses a substring of secret with correct length for comparison with guess</td>
<td>Responses can still earn the point even if they</td>
<td>1 point</td>
</tr>
<tr>
<td></td>
<td>• only call secret.indexOf(guess)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• use == instead of equals</td>
<td></td>
</tr>
<tr>
<td>3    Loops through all necessary substrings of secret (no bounds errors)</td>
<td>Responses will not earn the point if they skip overlapping occurrences</td>
<td>1 point</td>
</tr>
<tr>
<td>4    Counts number of identified occurrences of guess within secret <em>(in the context of a condition involving both secret and guess)</em></td>
<td>Responses can still earn the point even if they</td>
<td>1 point</td>
</tr>
<tr>
<td></td>
<td>• initialize count incorrectly or not at all</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• identify occurrences incorrectly</td>
<td></td>
</tr>
<tr>
<td>5    Calculates and returns correct final score (algorithm)</td>
<td>Responses will not earn the point if they</td>
<td>1 point</td>
</tr>
<tr>
<td></td>
<td>• initialize count incorrectly or not at all</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• fail to use a loop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• fail to compare guess to multiple substrings of secret</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• count the same matching substring more than once</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• use a changed or incorrect guess length when computing the score</td>
<td></td>
</tr>
</tbody>
</table>

Total for part (a) 5 points
## (b) findBetterGuess

<table>
<thead>
<tr>
<th>Scoring Criteria</th>
<th>Decision Rules</th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 6 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 point |  |
| 7 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 point |  |
| 8 Determines which of *guess1* and *guess2* is alphabetically greater | Responses **can** still earn the point even if they reverse the comparison  
- reimplement `compareTo` incorrectly  
- use result of `compareTo` as if boolean | 1 point |  |
| 9 Returns the identified *guess1* or *guess2* (*algorithm*) | Responses **can** still earn the point even if they  
- call `scoreGuess` incorrectly  
- compare strings incorrectly  
- reverse a comparison  
- omit either comparison  
- fail to return a guess in some case | 1 point |  |

**Total for part (b)** 4 points

**Question-specific penalties**

None

**Total for question 1** 9 points
public int scoreGuess(String guess)
{
    int count = 0;
    String mod = secret;
    while (mod, indexOf(guess) > 0) {
        count++;
        mod = mod, substring(mod, indexOf(guess) + 1);
    }
    return count + guess.length();
}
```csharp
public string FindBetterGuess(string guess1, string guess2)
{
    int g1 = ScoreGuess(guess1);
    int g2 = ScoreGuess(guess2);

    if (g1 > g2)
        return guess1;
    else if (g2 > g1)
        return guess2;
    else
    {
        if (guess1.CompareTo(guess2) > 0)
            return guess1;
        else
            return guess2;
    }
}
```
a) public int scoreGuess(String guess)
    {
        int count = 0;
        String temp;
        for(int x = 0; x < secret.length(); x++)
        {
            if(secret.indexOf(guess) >= 0)
            {
                count = count + 1;
                temp = secret.subString(secret.indexOf(guess));
            }
        }
        int score = 0;
        score = count + guess.length() * guess.length();
        return score;
    }
b) public string findBetterGuess (string guess1, string guess2) {
    string bguess;
    if (scoreGuess(guess1) > scoreGuess(guess2)) {
        bguess = guess1;
    } else if (scoreGuess(guess1) < scoreGuess(guess2)) {
        bguess = guess2;
    } else {
        if (guess1.compare2 < 0) {
            return guess2;
        } else {
            return guess1;
        }
    }
    return bguess;
}
a) public int scoreGuess (String guess) {
    int occurrences;
    if (secret.indexOf (guess) < 0) {
        scoreGuess = 0;
    }
    else if (secret.indexOf (guess) > 0) {
        secret = secret.substring (secret.indexOf (guess), occurrences + 1);
        return occurrences * guess.length () * guess.length ()
    }
}

b) public String findBetterGuess (String guess1, String guess2) {
    if (game . scoreGuess (guess1) > game . scoreGuess (guess2)) {
        findBetterGuess = guess1;
    }
    if (game . scoreGuess (guess2) > game . scoreGuess (guess1)) {
        findBetterGuess = guess2;
    }
    else if (game . scoreGuess (guess1) = game . scoreGuess (guess2) {
        if (guess1 . compareTo (guess2) > 0) {
            findBetterGuess = guess1;
        }
        if (guess1 . compareTo (guess2) < 0) {
            findBetterGuess = guess2;
        }
        else if (guess1 . compareTo (guess2) = 0) {
            findBetterGuess = guess1;
        }
    }
}
Question 1

Overview

This question tested the student’s ability to:

- Write program code to create objects of a class and call methods.
- Write program code to satisfy methods using expressions, conditional statements, and iterative statements.

More specifically, this question assessed the ability to use String objects, iterate through a range, call String methods, and use a method’s return value in a conditional expression.

In part (a) students were asked to loop through substrings of secret to determine whether there is an occurrence of the string guess within secret. Students accumulated a count of the number of occurrences of guess within secret. They were expected to initialize a numeric counter, iterate through all the substrings of secret, and update the counter. The students then had to calculate the return value, which is the product of their counter and the square of the length of guess.

In part (b) students were asked to compare the results of a method call using conditional statements. They needed to test which return value from two calls to scoreGuess was greater and return the parameter with the higher return value. The students also needed to perform an alphabetical comparison of the two parameters if the return values from the scoreGuess method calls were equal. They needed to return the correct string based on their comparisons.

Sample: 1A
Score: 8

In part (a) point 1 was earned by calling indexOf on mod, with guess as a parameter. The variable mod is initially a reference to secret and later contains substrings of secret. The point is earned because indexOf effectively does a comparison between secret and guess to determine the position of the first occurrence of guess in secret. The String mod can be modified without destroying the persistent data stored in secret. Point 2 was earned by calling indexOf(guess) on a reference to secret. Point 3 was earned by looping through all necessary substrings of mod by creating a substring that begins at the index of the found guess plus 1. Point 4 was earned by counting identified occurrences of guess within secret in the context of a condition and within a while loop.

In part (b) point 5 was not earned because the returned value is count * guess.length() instead of the product of count and the square of guess.length(), although the count was correctly computed. Note that using the dot instead of an asterisk is not, by itself, a problem; using common mathematical symbols such as • for multiplication is one of the minor errors for which no penalty is assessed. (See the "No Penalty" category on page 1 of the Scoring Guidelines for a complete list.) Point 6 was earned by the two correct calls to the scoreGuess method with guess1 and guess2 as parameters. Point 7 was earned by testing if g1, the response’s local variable storing the score of guess1, is greater than g2, the response’s local variable storing the score of guess2. Point 8 was earned because the compareTo method is called correctly to compare the two guesses, and its result is compared to zero. Point 9 was earned because the response returns the correctly identified guess1 or guess2 in all required cases.
Question 1 (continued)

Sample: 1B
Score: 6

In part (a) point 1 was earned by comparing `guess` to `secret` using `indexOf`. Point 2 was earned by comparing `guess` to `secret` using `indexOf`. Point 3 was not earned because the response does not loop through all necessary substrings of `secret`. The variable `temp` is assigned a substring in the loop, as in one common solution strategy, but the substring and starting index are taken from the original value of `secret` rather than from `temp`, so the same substring is compared repeatedly. Point 4 was earned by counting the number of identified occurrences of `guess` within `secret`. Point 5 was not earned because the algorithm adds `count` to the square of `guess.length()` instead of multiplying.

In part (b) point 6 was earned because the response calls the `scoreGuess` method correctly with `guess1` and `guess2` as parameters. Point 7 was earned because the response compares the return values of the two `scoreGuess` method calls. Point 8 was not earned because the method `compare2` does not exist. Point 9 was earned because each comparison returns the identified `guess1` or `guess2`. The incorrect comparison from point 8 does not affect point 9 because the implied logic of the alphabetical comparison is correct.

Sample: 1C
Score: 4

In part (a) point 1 was earned by comparing `guess` to `secret` using `indexOf`. Point 2 was earned by comparing `guess` to `secret` using `indexOf`. A penalty (-1) was applied because the response modifies the value of `secret`. Responses should not destroy persistent data (e.g., modifying a private instance variable). Point 3 was not earned because the response does not include a loop. Point 4 was earned because the response increments a counter within the context of a conditional involving `secret` and `guess`. Without a loop, the response can identify at most one occurrence of `guess` within `secret`, even if other occurrences exist. Point 5 was not earned because the response does not include a loop.

In part (b) point 6 was not earned because the response calls the `scoreGuess` method on `game`, which is an object or class other than `this`. Point 7 was earned by comparing the results of the `scoreGuess` method calls. Point 8 was earned by determining whether `guess1` or `guess2` is alphabetically greater. Point 9 was not earned because the response does not include a `return` statement.