

**Com S 227
Spring 2023
Miniassignment
100 points**

Due Date: Friday, March 31, 11:59 pm (midnight)
5% bonus for submitting 1 day early (by 11:59 pm March 30)
10% penalty for submitting 1 day late (by 11:59 pm April 1)
No submissions accepted after April 1, 11:59 pm

This assignment is to be done on your own. See the Academic Integrity policy in the syllabus, for details.

You will not be able to submit your work unless you have completed the *Academic Dishonesty policy questionnaire* on the Assignments page on Canvas. Please do this right away.

Note: This is a miniassignment and the grading is completely automated.

If you do not submit it correctly and we have to run it by hand, you will receive at most half credit.

Contents

Overview	2
ChatGPT.....	2
Problems	5
1.	5
2.	7
3.	9
4.	11
Advice.....	13
My code's not working!!.....	13
The SpecChecker	14
Style and documentation.....	15
If you have questions	15
What to turn in.....	16

Overview

This is a short set of practice problems involving writing loops. You will write four methods for the class `mini.LoopsInfinityAndBeyond`. All of the methods are static, so your class will not have any instance variables (or any static variables, for that matter). There is a constructor, but it is declared `private` so the class cannot be instantiated.

There is a skeleton of the class on Canvas. If you use the skeleton code, be sure you put it in a package called `mini`.

You do *not* need arrays or ArrayLists for this assignment. Although you will not be penalized for using them.

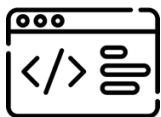
ChatGPT

For this assignment only, you are allowed to use ChatGPT. However, you are not required to create an account or use ChatGPT, the responses to all questions are provided in this document.

What is ChatGPT?

ChatGPT is an artificial intelligence (AI) chatbot. A user interacts with it by natural language queries and it responds in text. It works by using a model that has been trained with example answers to predict what an answer should look like. In some cases this produces correct answers and as we will see, it often produces very well written, convincing answers that are horribly wrong.

Here is an example interaction with ChatGPT, where it found information that was difficult to find with Google search. ChatGPT does a good job of summarizing the information, but we see a limitation is its inability to provide references for the information it presents. We should take its word that its answer is “widely recognized and accepted”?



Why is `< / >` image used to represent code?



The symbols `<` and `>` are commonly used to represent code in text-based communication because they resemble the angle brackets used in many programming languages to denote code blocks or tags.

For example, in HTML and XML, angle brackets are used to surround tags that specify how content should be displayed or structured on a web page. In programming languages like Java and C++, angle brackets are used to indicate parameterized types or generic classes.

Using `<` and `>` to represent code is also convenient because they are easy to type on a keyboard and are not commonly used in normal text communication. This makes it easy to visually distinguish code from other types of text.

Additionally, using angle brackets to represent code is a convention that has been used for many years in online discussion forums and other text-based communication platforms, so it has become widely recognized and understood by many people.



What is the origin of this usage?



The origin of using angle brackets `<` and `>` to represent code is not entirely clear, as it has been used in various contexts over the years.

One possibility is that it originated in online bulletin board systems (BBS) and Usenet newsgroups in the 1980s and 1990s, where users would discuss computer programming and share code snippets. In these forums, users would use angle brackets to indicate code blocks or tags, similar to how they are used in programming languages.

Another possibility is that the use of angle brackets to represent code was popularized by the markup language HTML, which was introduced in the early 1990s. HTML uses angle brackets to enclose tags that define how content should be displayed in a web browser, and this convention may have influenced the use of angle brackets to represent code in other contexts.

Regardless of its exact origins, the use of angle brackets to represent code has become a widely recognized convention in many online forums, messaging platforms, and programming communities.



What is the reference for this information?



The information I provided is based on my knowledge and experience as a language model trained by OpenAI, and from my research on the history of online communication and programming languages. While I cannot provide a specific reference for this information, it is widely recognized and accepted within the programming and online communication communities. The use of angle brackets to denote code blocks or tags is a convention that has been in use for many years in various contexts, and is widely recognized and understood by many people today.

ChatGPT can also solve programming problems and even provide documentation, explanations, and examples of how the code works. Here is an example of ChatGPT providing code and an explanation to a simple programming challenge.



Write a Java method that removes all vowels from a given string.



To remove all vowels from a given string, we can create a new string and iterate through the characters of the input string. For each character, we check if it is a vowel. If it's not a vowel, we append it to the new string. After the loop, we return the new string.

Here's the implementation of this algorithm in Java:

```
public static String removeVowels(String s) {  
    if (s == null) {  
        return null;  
    }  
    StringBuilder sb = new StringBuilder();  
    for (int i = 0; i < s.length(); i++) {  
        char c = s.charAt(i);  
        if (c != 'a' && c != 'e' && c != 'i' && c != 'o' && c != 'u'  
            && c != 'A' && c != 'E' && c != 'I' && c != 'O' && c != 'U') {  
            sb.append(c);  
        }  
    }  
    return sb.toString();  
}
```

We first handle the special case where the input string is null by returning null. Otherwise, we create a StringBuilder to store the output string, and we loop through the characters of the input string using an index *i*. For each character, we check if it is a vowel by comparing it to the vowels in both lowercase and uppercase. If it's not a vowel, we append it to the output string.

After the loop, we return the output string.

Here are some examples of using this method:

```
String s1 = "Hello, world!";  
String result1 = removeVowels(s1); // result1 = "Hll, wrld!"  
  
String s2 = "Java is awesome";  
String result2 = removeVowels(s2); // result2 = "Jv s wsm"  
  
String s3 = "";  
String result3 = removeVowels(s3); // result3 = ""
```

However, the code is very often buggy or even just completely wrong. ChatGPT has limited to no reasoning skills, it is good at creating something that looks like a correct solution based on its training from seeing the solutions to similar problems found on the Internet. But when a problem requires applying logic it often produces wrong code. As we will see, it even lies, claiming the code passes the example test cases when it does not.

Will ChatGPT and other AI tools be widely used for programming? It can give a good head start on some types of problems, but it can also be very misleading. For this reason, a key skill is the ability to test and check the correctness of code, something that will be practiced in this assignment. It's clear that for the time being ChatGPT cannot replace even beginning programmers. If you plan to work in software development, it is important to be aware of new tools and to know their capabilities and limitations. In this mini assignment we explore using ChatGPT to solve problems that require writing basic algorithms with loops.

Problems

In the following four problems you are expected to implement the specified static methods. You may use helper methods as long as they are private. The speccheck has some basic tests for the methods, however, additional tests will be added for grading. It is up to you to implement the methods according to the descriptions below and to perform your own testing.

1.

For this problem write a method with the following signature:

```
public static int flyingSaucerLength(String s) {
```



Define a flying saucer as the following string pattern: one '(', followed by zero to many '=', followed by one ')'.

Write a Java method that, given a string find the first instance of a flying saucer (starting from the left) and return its length. If no flying saucer exists return 0.

For example:

Given: "(==)"

Return: 4

Given: "*** () ** (==)"

Return: 2

Given: "***** (***)"

Return: 0

Do not use recursion or regular expressions (do not use the String method matches).



To solve this problem, we can iterate through the string from left to right, keeping track of the number of '=' characters seen so far. When we encounter an opening '(' character, we reset the count to zero. When we encounter a closing ')' character, we check if the count is greater than zero, indicating that we have seen at least one '=' character between the opening and closing parentheses. If so, we return the length of the substring between the parentheses.

If we reach the end of the string without finding a flying saucer, we return 0.

Here's the implementation of this algorithm in Java:

```
public static int flyingSaucerLength(String s) {  
    int count = 0;  
    int start = -1;  
    for (int i = 0; i < s.length(); i++) {  
        if (s.charAt(i) == '(') {  
            count = 0;  
            start = i;  
        } else if (s.charAt(i) == ')') {  
            if (count > 0) {  
                return i - start + 1;  
            }  
        } else if (s.charAt(i) == '=') {  
            count++;  
        }  
    }  
    return 0;  
}
```

We initialize **count** to zero and **start** to -1 to indicate that we haven't seen an opening parenthesis yet. We then loop through the characters of the string using an index **i**. For each character, we check if it's an opening parenthesis, closing parenthesis, or equals sign. If it's an opening parenthesis, we reset the count to zero and update the start index to the current position.

If it's a closing parenthesis, we check if the count is greater than zero. If it is, we return the length of the substring between the start index and the current position (inclusive of both parentheses).

If it's an equals sign, we increment the count.

If we reach the end of the string without finding a flying saucer, we return 0.

Here are some examples of using this method:

```
String s1 = "(=)";  
int result1 = findFlyingSaucer(s1); // result1 = 4  
  
String s2 = "****()***";
```

```
int result2 = findFlyingSaucer(s2); // result2 = 2

String s3 = "****(***)";
int result3 = findFlyingSaucer(s3); // result3 = 0
```

Commentary: The approach is reasonable and the explanation is convincing. Unfortunately, the solution does not work in all cases. Find where it doesn't work and fix the solution or create your own original solution to the problem.

2.

For this problem write a method with the following signature:

```
public static String fixFlyingSaucer(String s) {
```



Write a Java method that, given a string which may contain a flying saucer broken into two parts with characters in between, return a string where the flying is fixed by removing the in between characters. Look for the two parts of the flying saucer from left to right and fix the saucer with the first available parts.

For example:

Given: *** (==*****==) ***

Return: *** (=====) ***

Given: *** (==*****) **=) *

Return: *** (==) **=) *

Given: *** (==) **

Return: *** (==) **



To solve this problem, we can use the same approach as the previous problem, i.e., iterate through the string from left to right and keep track of the start and end indices of the first available flying saucer. Once we find the indices, we can construct the fixed saucer by concatenating the first part, the equals signs, and the second part.

Here's the implementation of this algorithm in Java:

```
public static String fixFlyingSaucer(String s) {
    int count = 0;
    int start = -1;
    int end = -1;
    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) == '(') {
            count = 0;
            start = i;
        } else if (s.charAt(i) == ')') {
```

```

        if (count > 0 && start != -1) {
            end = i;
            break;
        } else {
            start = -1;
        }
    } else if (s.charAt(i) == '=') {
        count++;
    }
}
if (start != -1 && end != -1) {
    String firstPart = s.substring(0, start);
    String secondPart = s.substring(end + 1);
    int numEquals = end - start - 1;
    String fixedSaucer = firstPart + "(" + "=".repeat(numEquals)
+ ")" + secondPart;
    return fixedSaucer;
} else {
    return s;
}
}

```

We initialize **count**, **start**, and **end** to -1 to indicate that we haven't found a flying saucer yet. We then loop through the characters of the string using an index **i**. For each character, we check if it's an opening parenthesis, closing parenthesis, or equals sign.

If it's an opening parenthesis, we reset the count to zero and update the start index to the current position.

If it's a closing parenthesis, we check if the count is greater than zero and if we have found a starting index. If both conditions are true, we update the end index to the current position and break out of the loop. Otherwise, we reset the start index to -1.

If it's an equals sign, we increment the count.

After the loop, if we have found a starting and ending index, we extract the first part of the string before the starting index, the second part of the string after the ending index, and the number of equals signs between the starting and ending indices. We then construct the fixed flying saucer by concatenating the first part, the opening parenthesis, the required number of equals signs, the closing parenthesis, and the second part.

If we haven't found a flying saucer, we simply return the input string.

Here are some examples of using this method:

```

String s1 = "***(=====)***";
String result1 = fixFlyingSaucer(s1); // result1 = ***(=====)***

String s2 = "***(=====)*=";
String result2 = fixFlyingSaucer(s2); // result2 = ***(=====)*

```



```
String s3 = "***(==)**";  
String result3 = fixFlyingSaucer(s3); // result3 = "***(==)**"
```

Commentary: Here is an interesting case where not only is ChatGPT's solution wrong, it is lying about its solution passing the example tests. For example, if you call the method with s2 you will see that the string returned is not the claimed result "***(==)**". This is something ChatGPT does often. Keep in mind that it works by predicting what a convincing answer should look at and not by actually running its own code or applying reasoning skills. At this point you may be inspired by ChatGPT's solution, but it may be better to just create your own approach to solve the problem.

3.

For this problem write a method with the following signature:

```
public static String countFlyingSaucers(String s) {
```



Write a Java method that, given a string which may contain many flying saucers, return the number of flying saucers. For this problem a flying saucer may wrap around from the right side of the string to the left.

For example:

Given: *** (==) ***

Return: 1

Given: =) ** (==) ** (

Return: 2

Given: *** (=*=) **

Return: 0



To solve this problem, we can use a similar approach as the first problem, i.e., iterate through the string from left to right and keep track of the number of opening parentheses and equals signs seen so far. When we encounter a closing ')' character, we check if there is a matching '(' character to form a flying saucer. If not, we check if there is a matching ')' character to form a wrapped flying saucer from the end of the string. If there is, we subtract the number of equals signs seen from the total count of characters and return the number of flying saucers found.

Here's the implementation of this algorithm in Java:

```
public static int countFlyingSaucers(String s) {  
    int count = 0;  
    int openCount = 0;  
    int equalCount = 0;  
    for (int i = 0; i < s.length(); i++) {
```

```

        if (s.charAt(i) == '(') {
            openCount++;
        } else if (s.charAt(i) == ')') {
            if (openCount > 0) {
                openCount--;
                count++;
            }
        } else if (s.charAt(i) == '=') {
            equalCount++;
        }
    }
    for (int i = s.length() - 1; i >= 0; i--) {
        if (s.charAt(i) == ')') {
            int wrappedEqualCount = s.length() - i - 2 - equalCount;
            if (openCount > 0 && wrappedEqualCount >= openCount) {
                count += openCount;
                openCount = 0;
            } else if (wrappedEqualCount >= 0) {
                equalCount += wrappedEqualCount;
            }
        } else if (s.charAt(i) == '(') {
            openCount--;
        } else if (s.charAt(i) == '=') {
            equalCount--;
        }
    }
    return count;
}

```

We initialize **count**, **openCount**, and **equalCount** to zero to indicate that we haven't found any flying saucers yet. We then loop through the characters of the string using an index **i**. For each character, we check if it's an opening parenthesis, closing parenthesis, or equals sign.

If it's an opening parenthesis, we increment the open count.

If it's a closing parenthesis, we check if there is a matching opening parenthesis. If there is, we decrement the open count and increment the flying saucer count.

If it's an equals sign, we increment the equal count.

After the loop, we loop through the characters of the string again from right to left using an index **i**. For each character, we check if it's a closing parenthesis, opening parenthesis, or equals sign.

If it's a closing parenthesis, we check if there is a matching opening parenthesis and if there are enough equals signs to form a wrapped flying saucer. If both conditions are true, we increment the flying saucer count by the number of matching opening parentheses and reset the open count to zero. If there are not enough equals signs, we add the number of equals signs to the total count of equals signs.

If it's an opening parenthesis, we decrement the open count.

If it's an equals sign, we decrement the equal count.

After the loop, we return the count of flying saucers found.

Here are some examples of using this method:

```
String s1 = "***(==)***";  
int result1 = countFlyingSaucers(s1); // result1 = 1  
  
String s2 = "=)**(==)**(";   
int result2 = countFlyingSaucers(s2); // result2 = 2
```

Commentary: As the problems becomes more complex, ChatGPT's solutions fail even more tests.

4.

For this problem write a method with the following signature:

```
public static String flyingSaucersFly(String s) {
```



Write a Java method that, given a string which may contain many flying saucers, shifts all of the saucers one character to the right. For this problem a flying saucer may wrap around from the right side of the string to the left. The returned string should have the same number of characters as the given string. This is achieved by moving the character to the right of a saucer to its left. It can be assumed that saucers will never be touching each other (i.e., there will always be at least one character between any two saucers).

For example:

Given: *** (==) ***

Return: **** (==) **

Given: =)** (==) ** (

Return: (=)** (==) **

Given: a () bcde (=*) fg

Return: ab () cde (=*) fg



To solve this problem, we can use a similar approach as the first problem, i.e., iterate through the string from left to right and keep track of the number of opening parentheses and equals signs seen so far. When we encounter a closing ')' character, we check if there is a matching '(' character to form a flying saucer. If so, we shift the next character to the right of the saucer to the left. If not, we check if there is a matching ')' character to form a wrapped flying saucer from the end of the string. If there is, we shift the

next character to the right of the saucer to the left. We repeat this process for all flying saucers found.

Here's the implementation of this algorithm in Java:

```
public static String flyingSaucersFly(String s) {
    char[] arr = s.toCharArray();
    int openCount = 0;
    int equalCount = 0;
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == '(') {
            openCount++;
        } else if (arr[i] == ')') {
            if (openCount > 0) {
                openCount--;
                int j = i + 1;
                while (j < arr.length && arr[j] == '=') {
                    char temp = arr[j - 1];
                    arr[j - 1] = arr[j];
                    arr[j] = temp;
                    j++;
                }
            }
        } else if (arr[i] == '=') {
            equalCount++;
        }
    }
    for (int i = arr.length - 1; i >= 0; i--) {
        if (arr[i] == ')') {
            int wrappedEqualCount = arr.length - i - 2 - equalCount;
            if (openCount > 0 && wrappedEqualCount >= openCount) {
                int j = i + 1;
                while (j < arr.length && arr[j] == '=') {
                    char temp = arr[j - 1];
                    arr[j - 1] = arr[j];
                    arr[j] = temp;
                    j++;
                }
                openCount--;
            } else if (wrappedEqualCount >= 0) {
                equalCount += wrappedEqualCount;
                char temp = arr[arr.length - 1];
                for (int k = arr.length - 1; k > i; k--) {
                    arr[k] = arr[k - 1];
                }
                arr[i + 1] = temp;
            }
        } else if (arr[i] == '(') {
            openCount--;
        } else if (arr[i] == '=') {
            equalCount--;
        }
    }
    return new String(arr);
}
```

```
}
```

We first convert the input string into a char array, which makes it easier to modify individual characters. We then initialize **openCount** and **equalCount** to zero to indicate that we haven't found any flying saucers yet. We then loop through the characters of the string using an index **i**. For each character, we check if it's an opening parenthesis, closing parenthesis, or equals sign.

If it's an opening parenthesis, we increment the open count.

If it's a closing parenthesis, we check if there is a matching opening parenthesis. If there is, we decrement the open count and shift the next character to the right of the saucer to the left.

If it's an equals sign, we increment the equal count.

After the loop, we loop through the characters of the string again from right to left using an index **i**. For each character, we check if it's a closing parenthesis, opening parenthesis, or equals sign

Commentary: It doesn't take looking any further than the two outer for loops to see that the ChatGTP solution is complete nonsense. Here we see that ChatGTP is not applying basic reasoning skills; there is no reasonable solution to the problem that would use the structure of two outer loops in sequence. When you develop your own solution, a better starting point would be one single outer loop that iterates through the entire string. The solution may also require multiple inner loops.

Advice

Before you write any code for a method, work through the problem with a pencil and paper on a few concrete examples. Make yourself write everything down; in particular, write down things that you need to remember from one step to the next (such as indices, or values from a previous step). Try to explain what you are doing in words. Write your algorithm in pseudocode. Explain it to your mom. Explain it to your dog. Find a bright fourth grader and explain it to her. *If you can't explain your algorithm so that a fourth grader can follow the steps, then you probably can't get the Java runtime to follow it either.*

The other key problem-solving strategy is to remember that you don't have to solve the whole problem in your head all at once. Try solving *part* of the problem, or solving a *related, simpler problem*.

My code's not working!!

Developing loops can be hard. Some of the problems in this assignment, although they are all short, are probably hard enough that if you don't have a clear idea of what you *want* the code to

do, you will be unable to successfully write code that works. You can waste many, many hours making random changes trying to get something to pass the sample tests. *Please don't do that.*

If you are getting errors, a good idea is to go back to a simple concrete example, describe your algorithm in words, and execute the steps by hand.

If your strategy works when you carry out the steps by hand, and you are confident that your algorithm is right but you are still getting errors, you then have a *debugging* problem – at some point you've coded something that isn't producing the result you intend.

In simple cases, you can verify what's happening in the code by temporarily inserting `println` statements to check whether variables are getting updated in the way you expect. (Remember to remove the extra `println`'s when you're done!)

Ultimately, however, the most powerful way to trace through code is with the debugger, as we are practicing in Lab 6. Learn to use the debugger effectively, and it will be a lifelong friend.

If you have an infinite loop, please refer to “Syllabus” -> “Technical Guides and Resources” -> “Tips for debugging infinite loops” on Canvas for additional tips.

The SpecChecker

A SpecChecker will posted with a number of functional tests. However, when you are debugging, it is usually helpful if you have a simpler test case of your own.

Remember that to call a static method, you prefix it with the *class* name, not with an object reference. For example, here is simple test case for the `doubleChars` method:

```
import mini.LoopsInfinityAndBeyond;
public class SimpleTest
{
    public static void main(String[] args)
    {
        String result = LoopsInfinityAndBeyond.flyingSaucerLength(" (==) ");
        System.out.println(result);
        System.out.println("Expected 4");
    }
}
```

You can save yourself from having to type "`LoopsInfinityAndBeyond`" over and over again by using the Java feature `import static`:

```
import static mini.LoopsInfinityAndBeyond.*;
public class LoopTester2
{
    public static void main(String[] args)
    {
        String result = flyingSaucerLength(" (==) ");
    }
}
```

```
        System.out.println(result);  
        System.out.println("Expected 4");  
    }  
}
```

Since no test code is being turned in, you are welcome to post your tests on Piazza for others to use and comment on.

Style and documentation

Since this is a miniassignment, the grading is automated and in most cases we will not be reading your code. Therefore, there are no specific documentation and style requirements. However, writing a brief descriptive comment for each method will help you clarify what it is you are trying to do. Likewise, brief internal comments can help you keep track of what you are trying to do when you write a tricky line of code.

If you have questions

For questions, please see the Piazza Q&A pages and click on the folder **miniassignment1**. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag **miniassignment1**. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in. (In the Piazza editor, use the button labeled “pre” to have Java code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post “private” so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form “read all my code and tell me what’s wrong with it” will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any posts from the instructors on Piazza that are labeled “Official Clarification” are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

What to turn in

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Homework page on Blackboard, before the submission link will be visible to you.

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **SUBMIT_THIS_mini.zip**. and it will be located in the directory you selected when you ran the SpecChecker. It should contain one directory, **mini**, which in turn contains one file, **LoopsInfinityAndBeyond.java**. Always LOOK in the zip file the file to check.

*We strongly recommend that you just submit the zip file created by the specchecker, AFTER CHECKING THAT IT CONTAINS THE CORRECT CODE. **If you mess something up and we have to run your code manually, you will receive at most half the points.***

Submit the zip file to Canvas using the Miniassignment1 submission link and verify that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO" which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links."

We strongly recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **mini**, which in turn should contain the file **LoopsInfinityAndBeyond.java**. You can accomplish this by zipping up the **src** directory of your project. The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and not a third-party installation of WinRAR, 7-zip, or Winzip.