709. To Lower Case

Easy Topics Companies Hint

Given a string `s`, return *the string after replacing every uppercase letter with the same lowercase letter*.

Example 1:

```
Input: s = "Hello"
Output: "hello"
```

Example 2:

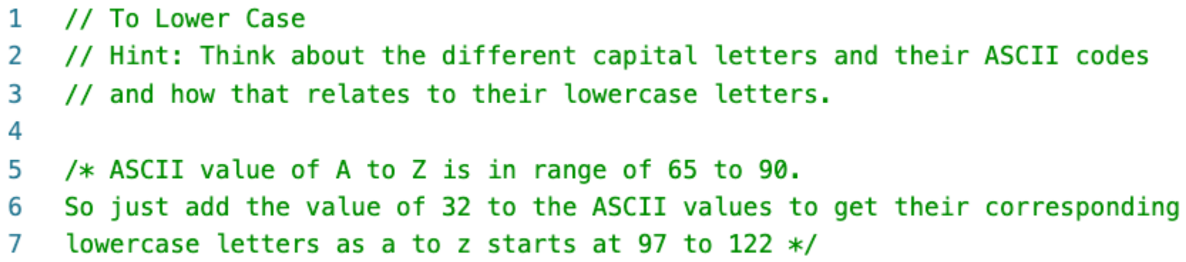
```
Input: s = "here"
Output: "here"
```

Example 3:

```
Input: s = "LOVELY"
Output: "lovely"
```

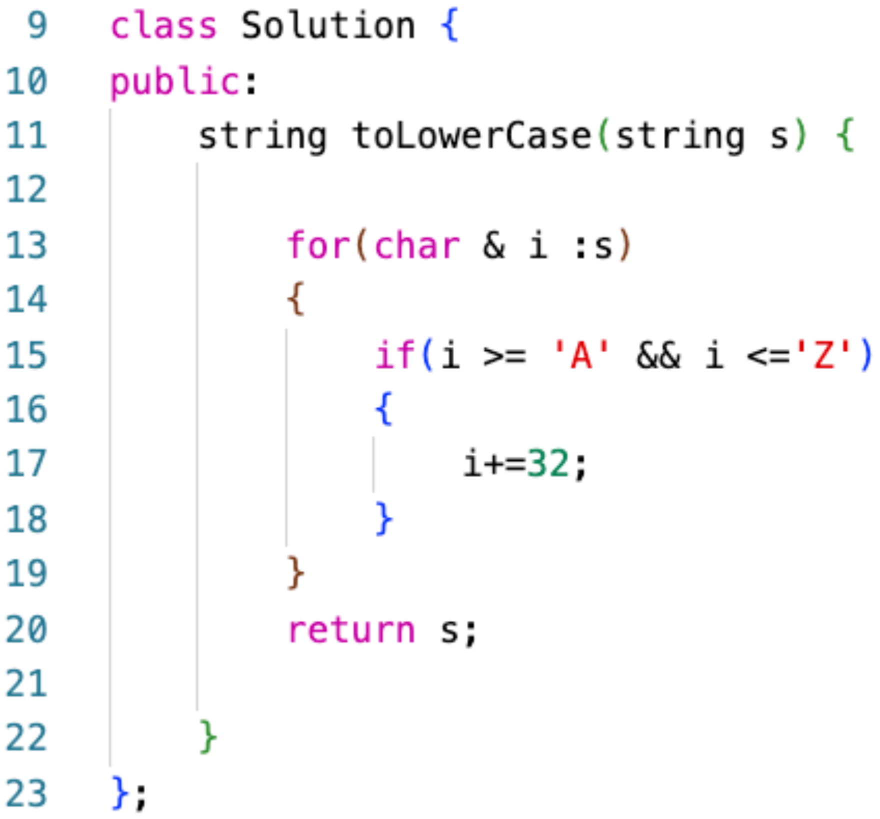
Constraints:

- `1 <= s.length <= 100`
- `s` consists of printable ASCII characters.



```
// To Lower Case
// Hint: Think about the different capital letters and their ASCII codes
// and how that relates to their lowercase letters.

/* ASCII value of A to Z is in range of 65 to 90.
So just add the value of 32 to the ASCII values to get their corresponding
lowercase letters as a to z starts at 97 to 122 */
```



```
class Solution {
public:
    string toLowerCase(string s) {

        for(char & i :s)
        {
            if(i >= 'A' && i <='Z')
            {
                i+=32;
            }
        }
        return s;

    }
};
```

771. Jewels and Stones

Easy Topics Companies Hint

You're given strings `jewels` representing the types of stones that are jewels, and `stones` representing the stones you have. Each character in `stones` is a type of stone you have. You want to know how many of the stones you have are also jewels.

Letters are case sensitive, so `"a"` is considered a different type of stone from `"A"`.

Example 1:

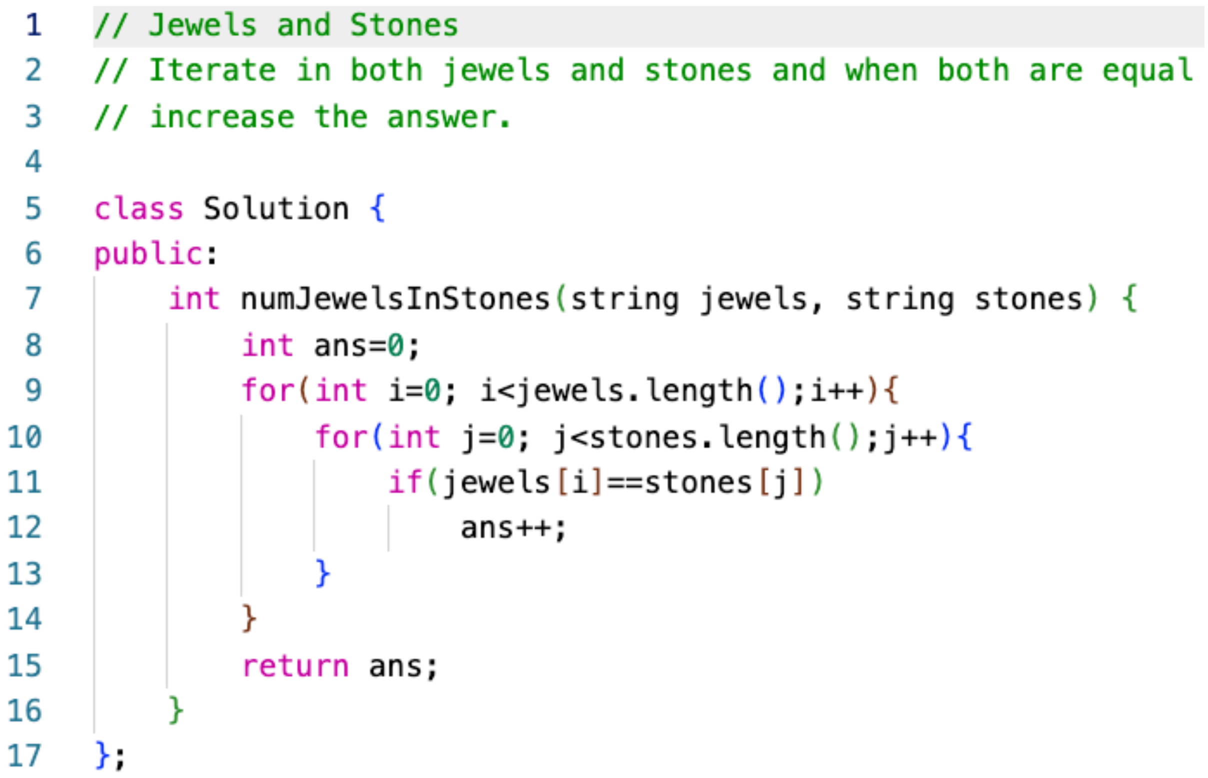
```
Input: jewels = "aA", stones = "aAAbbbb"
Output: 3
```

Example 2:

```
Input: jewels = "z", stones = "ZZ"
Output: 0
```

Constraints:

- `1 <= jewels.length, stones.length <= 50`
- `jewels` and `stones` consist of only English letters.
- All the characters of `jewels` are **unique**.



```
// Jewels and Stones
// Iterate in both jewels and stones and when both are equal
// increase the answer.

class Solution {
public:
    int numJewelsInStones(string jewels, string stones) {
        int ans=0;
        for(int i=0; i<jewels.length();i++){
            for(int j=0; j<stones.length();j++){
                if(jewels[i]==stones[j])
                    ans++;
            }
        }
        return ans;
    }
};
```

1544. Make The String Great

Easy · Topics · Companies · Hint

Given a string `s` of lower and upper case English letters.

A good string is a string which doesn't have **two adjacent characters** `s[i]` and `s[i + 1]` where:

- `0 <= i <= s.length - 2`
- `s[i]` is a lower-case letter and `s[i + 1]` is the same letter but in upper-case or **vice-versa**.

To make the string good, you can choose **two adjacent** characters that make the string bad and remove them. You can keep doing this until the string becomes good.

Return *the string* after making it good. The answer is guaranteed to be unique under the given constraints.

Notice that an empty string is also good.

Example 1:

```
Input: s = "leEeetcode"
Output: "leetcode"
Explanation: In the first step, either you choose i = 1 or i = 2, both will
result "leEeetcode" to be reduced to "leetcode".
```

Example 2:

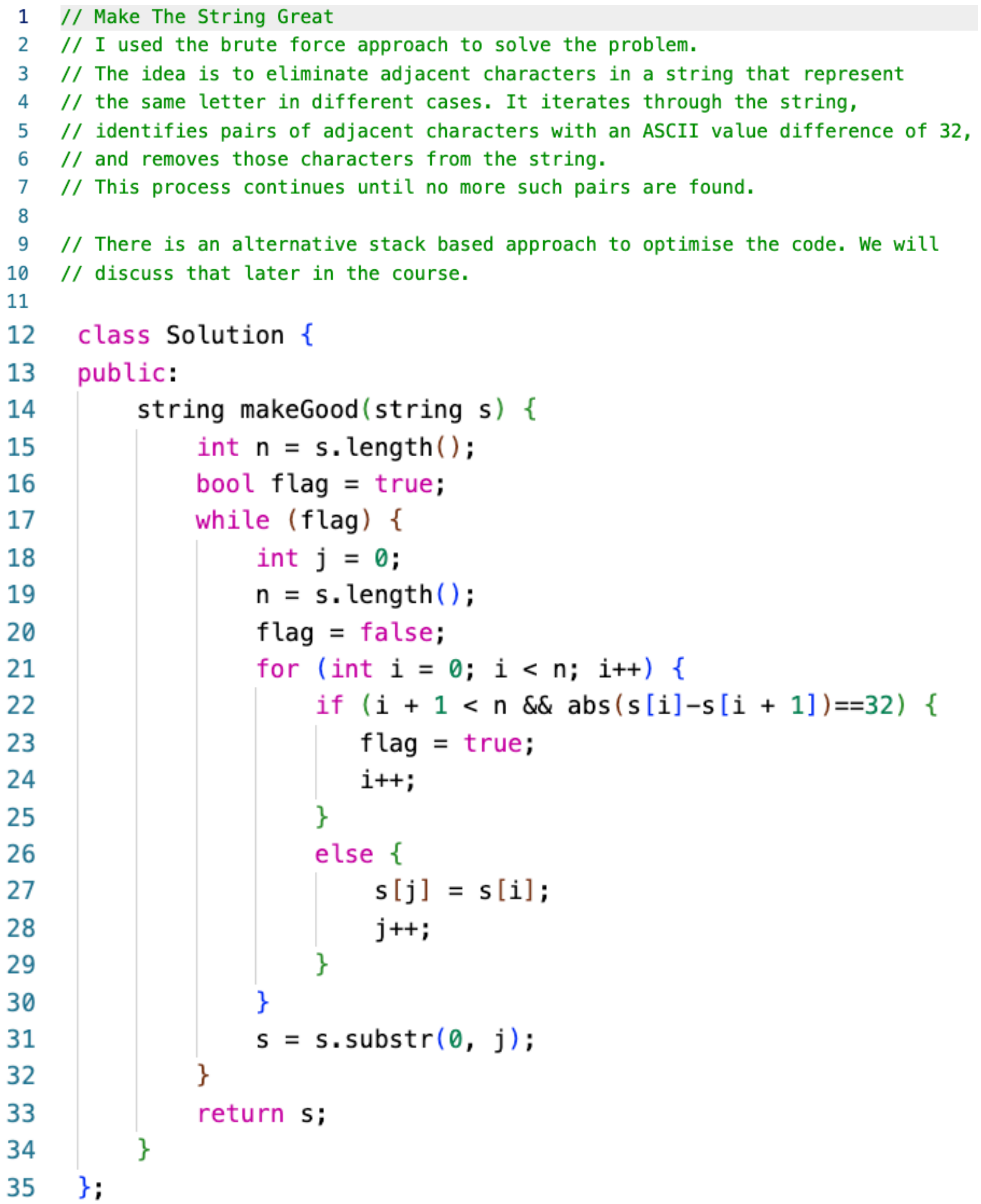
```
Input: s = "abBAcC"
Output: ""
Explanation: We have many possible scenarios, and all lead to the same
answer. For example:
"abBAcC" --> "aAcC" --> "cC" --> ""
"abBAcC" --> "abBA" --> "aA" --> ""
```

Example 3:

```
Input: s = "s"
Output: "s"
```

Constraints:

- `1 <= s.length <= 100`
- `s` contains only lower and upper case English letters.



```
// Make The String Great
// I used the brute force approach to solve the problem.
// The idea is to eliminate adjacent characters in a string that represent
// the same letter in different cases. It iterates through the string,
// identifies pairs of adjacent characters with an ASCII value difference of 32,
// and removes those characters from the string.
// This process continues until no more such pairs are found.

// There is an alternative stack based approach to optimise the code. We will
// discuss that later in the course.

 class Solution {
 public:
     string makeGood(string s) {
         int n = s.length();
         bool flag = true;
         while (flag) {
             int j = 0;
             n = s.length();
             flag = false;
             for (int i = 0; i < n; i++) {
                 if (i + 1 < n && abs(s[i]-s[i + 1])==32) {
                    flag = true;
                    i++;
                 }
                 else {
                     s[j] = s[i];
                     j++;
                 }
             }
             s = s.substr(0, j);
         }
         return s;
     }
 };
```

2315. Count Asterisks

Easy Topics 🔒 Companies Hint

You are given a string `s`, where every **two** consecutive vertical bars `'|'` are grouped into a **pair**. In other words, the 1st and 2nd `'|'` make a pair, the 3rd and 4th `'|'` make a pair, and so forth.

Return *the number of* `'*'` *in* `s`*,* ***excluding*** *the* `'*'` *between each pair of* `'|'`.

Note that each `'|'` will belong to **exactly** one pair.

Example 1:

```
Input: s = "l|*e*et|c**o|*de|"
Output: 2
Explanation: The considered characters are underlined: "l|*e*et|c**o|*de|".
The characters between the first and second '|' are excluded from the
answer.
Also, the characters between the third and fourth '|' are excluded from the
answer.
There are 2 asterisks considered. Therefore, we return 2.
```

Example 2:

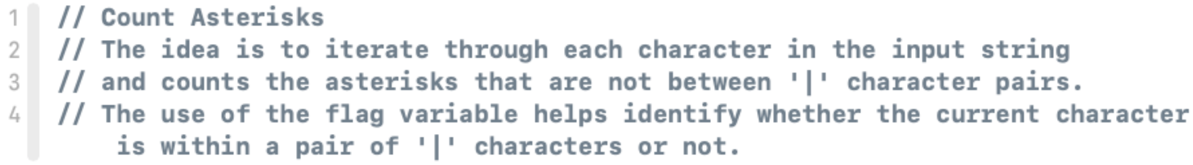
```
Input: s = "iamprogrammer"
Output: 0
Explanation: In this example, there are no asterisks in s. Therefore, we
return 0.
```

Example 3:

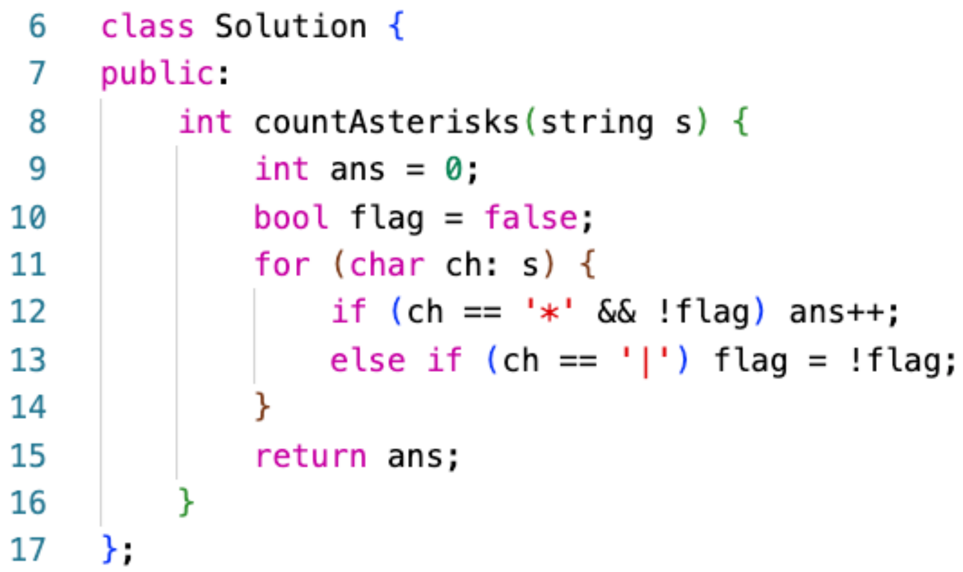
```
Input: s = "yo|uar|e**|b|e***au|tifu|l"
Output: 5
Explanation: The considered characters are underlined:
"yo|uar|e**|b|e***au|tifu|l". There are 5 asterisks considered. Therefore,
we return 5.
```

Constraints:

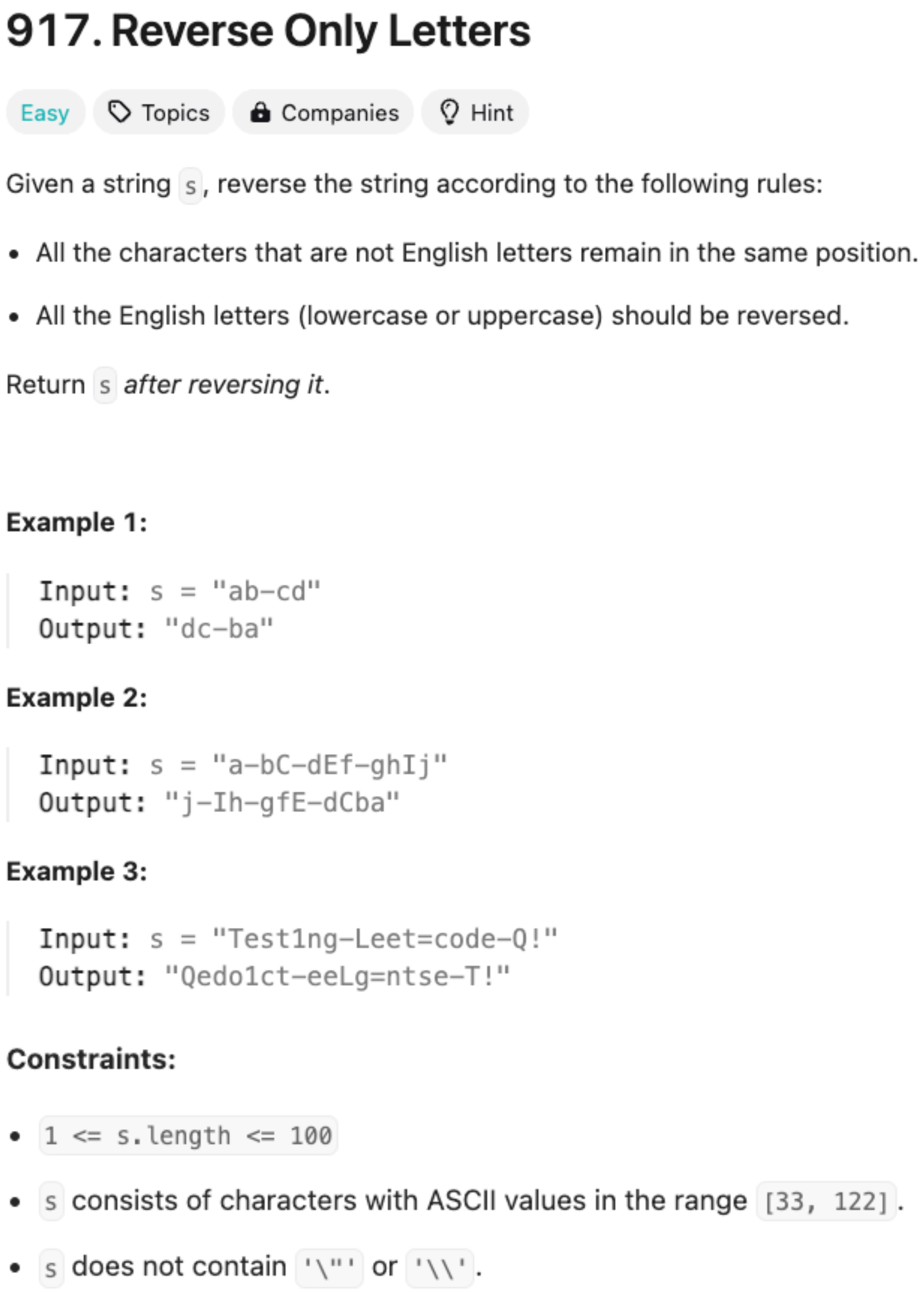
- `1 <= s.length <= 1000`
- `s` consists of lowercase English letters, vertical bars `'|'`, and asterisks `'*'`.
- `s` contains an **even** number of vertical bars `'|'`.



```
// Count Asterisks
// The idea is to iterate through each character in the input string
// and counts the asterisks that are not between '|' character pairs.
// The use of the flag variable helps identify whether the current character is within a pair of '|' characters or not.
```



```cpp
class Solution {
public:
    int countAsterisks(string s) {
        int ans = 0;
        bool flag = false;
        for (char ch: s) {
            if (ch == '*' && !flag) ans++;
            else if (ch == '|') flag = !flag;
        }
        return ans;
    }
};
```



917. Reverse Only Letters

Given a string `s`, reverse the string according to the following rules:

- All the characters that are not English letters remain in the same position.
- All the English letters (lowercase or uppercase) should be reversed.

Return `s` *after reversing it*.

Example 1:

```
Input: s = "ab-cd"
Output: "dc-ba"
```

Example 2:

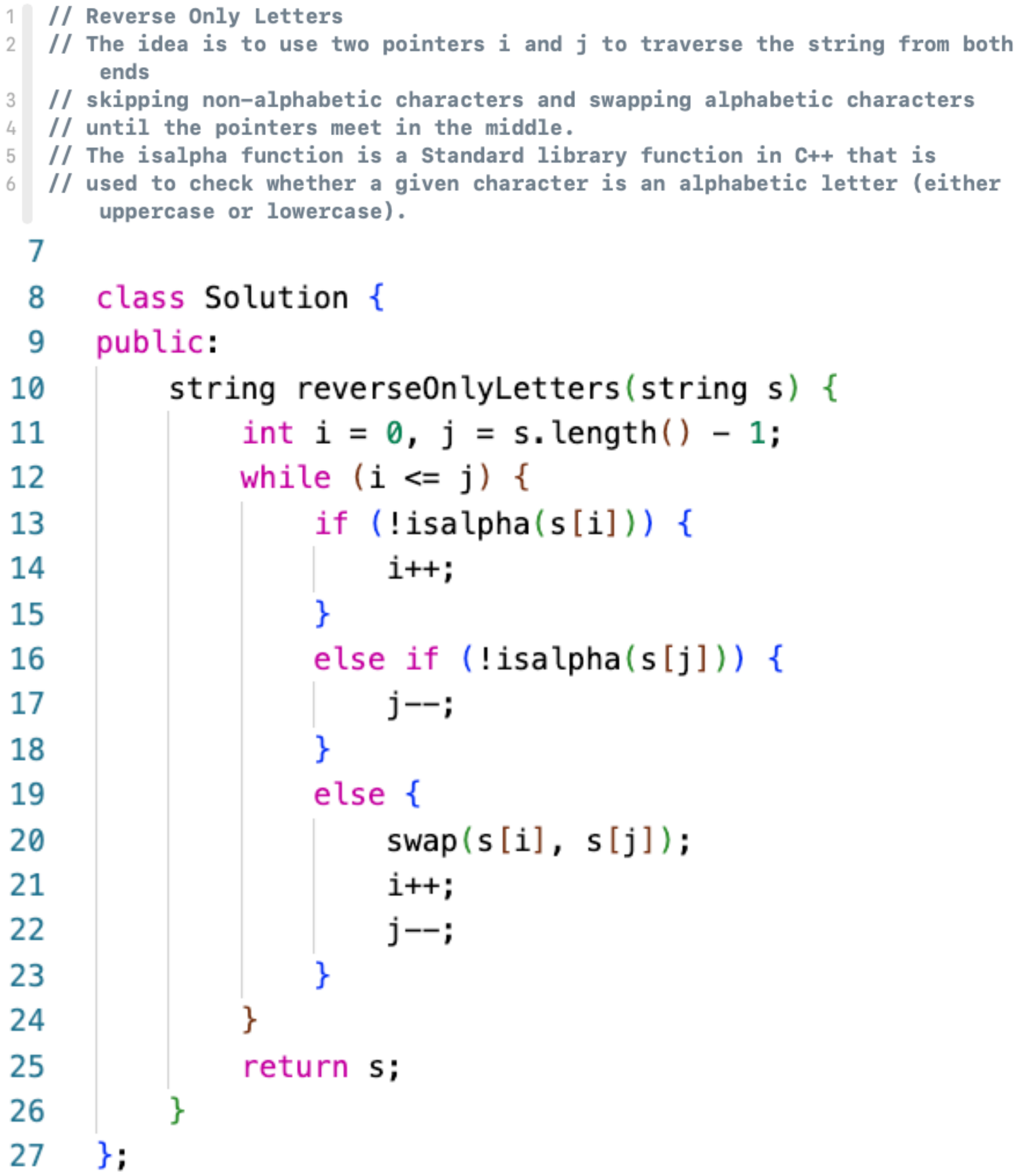
```
Input: s = "a-bC-dEf-ghIj"
Output: "j-Ih-gfE-dCba"
```

Example 3:

```
Input: s = "Test1ng-Leet=code-Q!"
Output: "Qedo1ct-eeLg=ntse-T!"
```

Constraints:

- `1 <= s.length <= 100`
- `s` consists of characters with ASCII values in the range `[33, 122]`.
- `s` does not contain `'\"'` or `'\\'`.



```cpp
// Reverse Only Letters
// The idea is to use two pointers i and j to traverse the string from both ends
// skipping non-alphabetic characters and swapping alphabetic characters
// until the pointers meet in the middle.
// The isalpha function is a Standard library function in C++ that is
// used to check whether a given character is an alphabetic letter (either uppercase or lowercase).

class Solution {
public:
    string reverseOnlyLetters(string s) {
        int i = 0, j = s.length() - 1;
        while (i <= j) {
            if (!isalpha(s[i])) {
                i++;
            }
            else if (!isalpha(s[j])) {
                j--;
            }
            else {
                swap(s[i], s[j]);
                i++;
                j--;
            }
        }
        return s;
    }
};
```

242. Valid Anagram

Easy Topics Companies

Given two strings `s` and `t`, return `true` *if* `t` *is an anagram of* `s`*, and* `false` *otherwise*.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example 1:

```
Input: s = "anagram", t = "nagaram"
Output: true
```

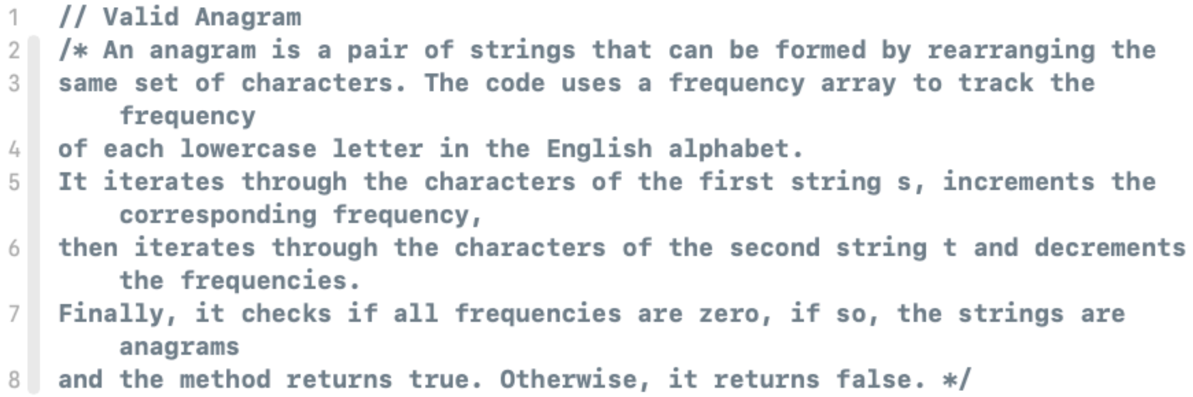
Example 2:

```
Input: s = "rat", t = "car"
Output: false
```

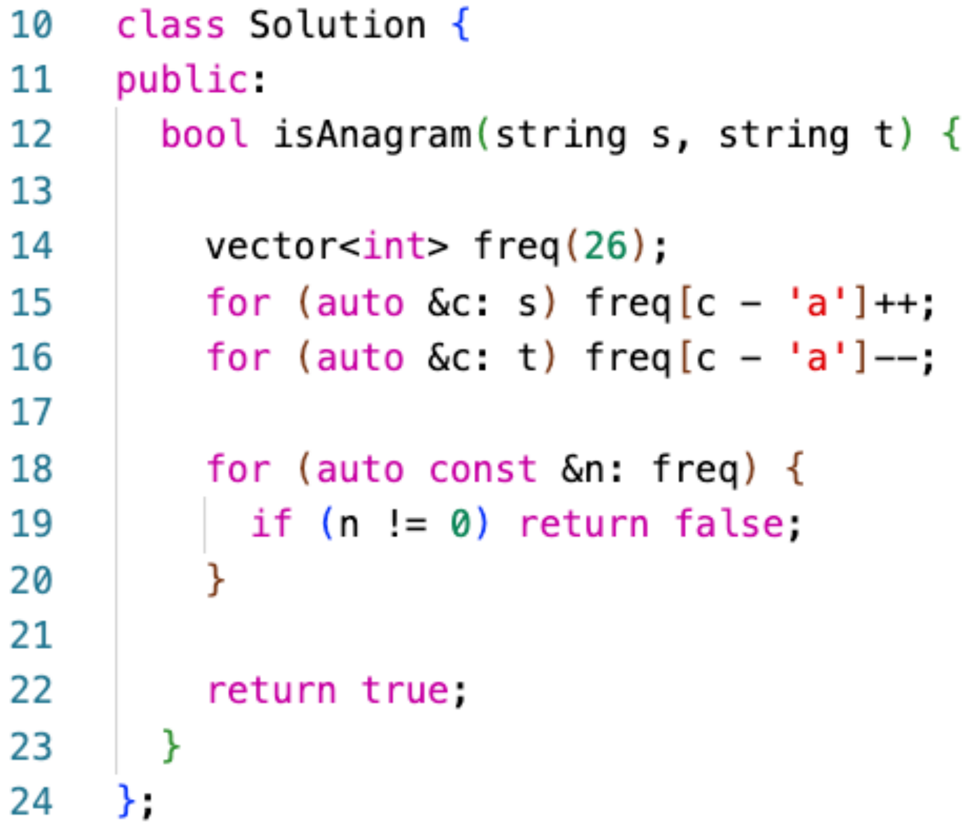
Constraints:

- `1 <= s.length, t.length <=` $5 * 10^4$
- `s` and `t` consist of lowercase English letters.

Follow up: What if the inputs contain Unicode characters? How would you adapt your solution to such a case?



```
// Valid Anagram
/* An anagram is a pair of strings that can be formed by rearranging the
same set of characters. The code uses a frequency array to track the frequency
of each lowercase letter in the English alphabet.
It iterates through the characters of the first string s, increments the corresponding frequency,
then iterates through the characters of the second string t and decrements the frequencies.
Finally, it checks if all frequencies are zero, if so, the strings are anagrams
and the method returns true. Otherwise, it returns false. */
```



```cpp
class Solution {
public:
  bool isAnagram(string s, string t) {

    vector<int> freq(26);
    for (auto &c: s) freq[c - 'a']++;
    for (auto &c: t) freq[c - 'a']--;

    for (auto const &n: freq) {
      if (n != 0) return false;
    }

    return true;
  }
};
```