# Question

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**.

# Solution

**Intuition and Algorithm**

For each stone, check whether it matches any of the jewels. We can check with a linear scan.

|  |
| --- |
| class Solution {  public int numJewelsInStones(String J, String S) {  int ans = 0;  for (char s: S.toCharArray()) // For each stone...  for (char j: J.toCharArray()) // For each jewel...  if (j == s) { // If the stone is a jewel...  ans++;  break; // Stop searching whether this stone 's' is a jewel  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(J\text{.length} \* S\text{.length}))*O*(*J*.length∗*S*.length)).
* Space Complexity: O(1)*O*(1) additional space complexity in Python. In Java, this can be O(J\text{.length} \* S\text{.length}))*O*(*J*.length∗*S*.length)) because of the creation of new arrays.

#### **Approach #2: Hash Set [Accepted]**

**Intuition and Algorithm**

For each stone, check whether it matches any of the jewels. We can check efficiently with a Hash Set.

|  |
| --- |
| class Solution {  public int numJewelsInStones(String J, String S) {  Set<Character> Jset = new HashSet();  for (char j: J.toCharArray())  Jset.add(j);  int ans = 0;  for (char s: S.toCharArray())  if (Jset.contains(s))  ans++;  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(J\text{.length} + S\text{.length})*O*(*J*.length+*S*.length). The O(J\text{.length})*O*(*J*.length) part comes from creating J. The O(S\text{.length})*O*(*S*.length) part comes from searching S.
* Space Complexity: O(J\text{.length})*O*(*J*.length).