Class Notes 8

- ▶ Python
- ▶ JavaScript
- ▼ Java

Question 1

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

Solution:

Intuition and Algorithm

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

Complexity Analysis

- Time Complexity: *O*(*J*.length+*S*.length). The *O*(*J*.length) part comes from creating J. The *O*(*S*.length) part comes from searching S.
- Space Complexity: *O*(*J*.length).

```
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;
    }
}
```

Question 2

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

Solution:

Algorithm

To examine if t is a rearrangement of s, we can count occurrences of each letter in the two strings and compare them. We could use a hash table to count the frequency of each letter, however, since both s and t only contain letters from a to z, a simple array of size 26 will suffice.

Do we need *two* counters for comparison? Actually no, because we can increment the count for each letter in s and decrement the count for each letter in t, and then check if the count for every character is zero.

Complexity Analysis

- Time complexity: *O*(*n*) because accessing the counter table is a constant time operation.
- Space complexity: *O*(1). Although we do use extra space, the space complexity is *O*(1) because the table's size stays constant no matter how large *n* is.

```
class Solution {
    public boolean isAnagram(String s, String t) {
      if (s.length() != t.length()) {
          return false;
      }
      int[] counter = new int[26];
      for (int i = 0; i < s.length(); i++) {
          counter[s.charAt(i) - 'a']++;
          counter[t.charAt(i) - 'a']--;
      }
      for (int count : counter) {
          if (count != 0) {
              return false;
          }
      }
```

```
        return true;
      }
    }
```

Question 3

A phrase is a palindrome if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string s, return true if it is a palindrome, or false otherwise.

Example 1:

Input: s = "A man, a plan, a canal: Panama"

Output: true

Explanation: "amanaplanacanalpanama" is a palindrome.

Solution:

Since the input string contains characters that we need to ignore in our palindromic check, it becomes tedious to figure out the real middle point of our palindromic input.

So, if we start traversing inwards, from both ends of the input string, we can expect to *see* the same characters, in the same order.

The resulting algorithm is simple:

- Set two pointers, one at each end of the input string
- If the input is palindromic, both the pointers should point to equivalent characters, *at all times*. 1
  - If this condition is not met at any point of time, we break and return early. 2
- We can simply ignore non-alphanumeric characters by continuing to traverse further.
- Continue traversing inwards until the pointers meet in the middle.

Complexity Analysis

- Time complexity : $O(n)$, in length n of the string. We traverse over each character at-most once, until the two pointers meet in the middle, or when we break and return early.
- Space complexity : $O(1)$. No extra space required, at all.

```
s Solution {
blic boolean isPalindrome(String s) {
for (int i = 0, j = s.length() - 1; i < j; i++, j--) {
  while (i < j && !Character.isLetterOrDigit(s.charAt(i))) {
    i++;
  }
  while (i < j && !Character.isLetterOrDigit(s.charAt(j))) {
    j--;
  }
  if (Character.toLowerCase(s.charAt(i)) != Character.toLowerCase(s.charAt(j
    return false;
}
return true;
```

Question 4

You are given an array of strings words (**0-indexed**).

In one operation, pick two **distinct** indices i and j, where words[i] is a non-empty string, and move **any** character from words[i] to **any** position in words[j].

Return true *if you can make **every** string in* words ***equal** using **any** number of operations*, *and* false *otherwise*.

Example 1:

Input: words = ["abc","aabc","bc"]

Output: true

Explanation: Move the first 'a' in words[1] to the front of words[2],

to make words[1] = "abc" and words[2] = "abc".

All the strings are now equal to "abc", so return true.

Solution:

Complexity Analysis

Time Complexity: O(N)

Space Complexity : O(1)

```
class Solution {
    public boolean makeEqual(String[] words) {
        int n = words.length;

        //26 word array to count each word frequency present in array
            int[] array = new int[26];

            for(String word : words){
                for(char c : word.toCharArray()){
                    array[c-'a']++;
```

```
            }
            for(int fq : array){
                if(fq%n != 0){
                    return false;
                }
            }
            return true;
        }
    }
```

Question 5

Balanced strings are those that have an equal quantity of 'L' and 'R' characters.

Given a **balanced** string s, split it into some number of substrings such that:

- Each substring is balanced.

Return *the **maximum** number of balanced strings you can obtain.*

Example 1:

Input: s = "RLRRLLRLRL"

Output: 4

Explanation: s can be split into "RL", "RRLL", "RL", "RL", each substring contains same number of 'L' and 'R'.

Solution:

Greedily split the string, and with the counting L +1

R -1, when the counter is reset to 0, we get one balanced string.

Complexity Analysis

Time Complexity : O(N)

Space Complexity : O(1)

```
class Solution {
  public int balancedStringSplit(String s) {
    int res = 0, cnt = 0;
    for (int i = 0; i < s.length(); ++i) {
        cnt += s.charAt(i) == 'L' ? 1 : -1;
        if (cnt == 0) ++res;
    }
    return res;
  }
}
```

Question 6

Given a string s, reverse only all the vowels in the string and return it.

The vowels are 'a', 'e', 'i', 'o', and 'u', and they can appear in both lower and upper cases, more than once.

Example 1:

Input: s = "hello"

Output: "holle"

Solution:

Algorithm

1. Initialize the left pointer start to 0, and the right pointer end to s.size() - 1.
2. Keep iterating until the left pointer catches up with the right pointer:
    a. Keep incrementing the left pointer start until it's pointing to a vowel character.
    b. Keep decrementing the right pointer end until it's pointing to a vowel character.
    c. Swap the characters at the start and end.
    d. Increment the start pointer and decrement the end pointer.
3. Return the string s.

Complexity Analysis

Here, *N* is the length of the string s.

- Time complexity: *O*(*N*)

  It might be tempting to say that there are two nested loops and hence the complexity would be

- *O*(*N*^2). However, if we observe closely the pointers start and end will only traverse the index once. Each element of the string s will be iterated only once either by the left or right pointer and not both. We swap characters when both pointers point to vowels which are *O*(1) operation. Hence the total time complexity will be *O*(*N*).

  Note that in Java we need to convert the string to a char array as strings are immutable and hence it would take *O*(*N*) time.

- Space complexity: *O*(*N*)

  In C++ we only need an extra temporary variable to perform the swap and hence the space complexity is *O*(1). However, in Java, we need to convert the string to a char array that would take *O*(*N*) space, and therefore the space complexity for Java would be *O*(*N*).

Java would be O(n).

```java
class Solution {
    // Return true if the character is a vowel (case-insensitive)
    boolean isVowel(char c) {
        return c == 'a' || c == 'i' || c == 'e' || c == 'o' || c == 'u'
            || c == 'A' || c == 'I' || c == 'E' || c == 'O' || c == 'U';
    }

    // Function to swap characters at index x and y
    void swap(char[] chars, int x, int y) {
        char temp = chars[x];
        chars[x] = chars[y];
        chars[y] = temp;
    }

    public String reverseVowels(String s) {
        int start = 0;
        int end = s.length() - 1;
        // Convert String to char array as String is immutable in Java
        char[] sChar = s.toCharArray();

        // While we still have characters to traverse
        while (start < end) {
            // Find the leftmost vowel
            while (start < s.length () && !isVowel(sChar[start])) {
                start++;
            }
            // Find the rightmost vowel
            while (end >= 0 && !isVowel(sChar[end])) {
                end--;
            }
            // Swap them if start is left of end
            if (start < end) {
                swap(sChar, start++, end--);
            }
        }

        // Converting char array back to String
        return new String(sChar);
    }
};
```