Practice Set II

More Contents on GitHub.com/RaSDE

28. Implement strStr()

Given two strings `needle` and `haystack`, return the index of the first occurrence of `needle` in `haystack`, or `-1` if `needle` is not part of `haystack`.

Clarification:
What should we return when `needle` is an empty string? This is a great question to ask during an interview.

For the purpose of this problem, we will return 0 when `needle` is an empty string. This is consistent to C's strstr() and Java's indexOf().

Example 1:

```
Input: haystack = "hello", needle = "ll"

Output: 2
```

Example 2:

```
Input: haystack = "aaaaa", needle = "bba"

Output: -1
```

1st Approach

```
class Solution {
    public int strStr(String haystack, String needle) {
        // Inbuild String Method
        return haystack.indexOf(needle);
    }
}
```

2nd Approach

```
class Solution {
    public int strStr(String haystack, String needle) {
        // Full Approach
        if(needle==null || needle.length()==0){
          return -1;
        }
        for(int i=0; ; i++){
            for(int j=0; ; j++){
            // if j becomes needle length, then the i is the starting index
                if(j==needle.length()) return i;
            // if i+j reaches at full length of haystack, then there is no matching index
                if(i+j == haystack.length()) return -1;
            // loop will break, if no matching chars are iterating
                if(haystack.charAt(i+j)!=needle.charAt(j)) break;
            }
        }
    }
}
```

459. Repeated Substring Pattern

Given a string `s`, check if it can be constructed by taking a substring of it and appending multiple copies of the substring together.

Example 1:

```
Input: s = "abab"

Output: true

Explanation: It is the substring "ab" twice.
```

Example 2:

```
Input: s = "aba"

Output: false
```

Example 3:

```
Input: s = "abcabcabcabc"

Output: true

Explanation: It is the substring "abc" four times or the substring "abcabc" twice.
```

```java
class Solution {
    public boolean repeatedSubstringPattern(String s) {
        // 1st Approach
        String str = s + s;    // abab + abab --> abababab | double the string
        // bababa | remove the first and last character
        str = str.substring(1, str.length()-1);
        // if there is repeatation,
        // then the new string must contains the original string
        return str.contains(s);
    }
}
```

```java
class Solution {
    public boolean repeatedSubstringPattern(String s) {
        // 2nd Approach
        int len = s.length();
        for(int i=len/2; i>=1; i--){
            if(len%i==0){
                int m = len/i;
                String sub = s.substring(0, i);
                StringBuilder sb = new StringBuilder();
                for(int j=0; j<m; j++){
                    sb.append(sub);
                }
                if(sb.toString().equals(s)) return true;
            }
        }
        return false;
    }
}
```

686. Repeated String Match

Given two strings `a` and `b`, return *the minimum number of times you should repeat string* `a` *so that string* `b` *is a substring of it*. If it is impossible for `b` to be a substring of `a` after repeating it, return `-1`.

Notice: string `"abc"` repeated 0 times is `""`, repeated 1 time is `"abc"` and repeated 2 times is `"abcabc"`.

Example 1:

```
Input: a = "abcd", b = "cdabcdab"

Output: 3

Explanation: We return 3 because by repeating a three times "abcdabcdabcd", b is a substring of
it.
```

Example 2:

```
Input: a = "a", b = "aa"

Output: 2
```

```
class Solution {
    public int repeatedStringMatch(String a, String b) {
        StringBuilder str = new StringBuilder();
        int count=0;
        while(str.length()<b.length()){
            str.append(a);
            count++;
        }
        if(str.toString().contains(b)){
                return count;
            }
        if(str.append(a).toString().contains(b)){
                return ++count;
            }
        return -1;
    }
}
```

1047. Remove All Adjacent Duplicates In String

You are given a string `s` consisting of lowercase English letters. A **duplicate removal** consists of choosing two **adjacent** and **equal** letters and removing them. We repeatedly make **duplicate removals** on `s` until we no longer can.Return *the final string after all such duplicate removals have been made*. It can be proven that the answer is **unique**.

Example 1:

```
Input: s = "abbaca"

Output: "ca"

Explanation:
For example, in "abbaca" we could remove "bb" since the letters are adjacent and equal, and
this is the only possible move.  The result of this move is that the string is "aaca", of which
only "aa" is possible, so the final string is "ca".
```

Example 2:

```
Input: s = "azxxzy"

Output: "ay"
```

```java
class Solution {
    public String removeDuplicates(String s) {
    // 1st Approach
        int i=0, n=s.length(), j=0;
        char[] arr = s.toCharArray();
        for(i=0; i<n; ++i, ++j){
            arr[j]=arr[i];
            if(j>0 && (arr[j-1]==arr[j])){
                j-=2;
            }
        }
        return String.copyValueOf(arr, 0, j);
    }
}
```

```java
class Solution {
    public String removeDuplicates(String s) {
    // 2nd Approach
        StringBuffer sb = new StringBuffer();
        for(char c : s.toCharArray()){
            int size=sb.length();
            if(size>0 && sb.charAt(size-1)==c){
                sb.deleteCharAt(size-1);
            }
            else
                sb.append(c);
        }
        return sb.toString();
    }
}
```

1209. Remove All Adjacent Duplicates in String II

You are given a string `s` and an integer `k`, a `k` **duplicate removal** consists of choosing `k` adjacent and equal letters from `s` and removing them, causing the left and the right side of the deleted substring to concatenate together.

We repeatedly make `k` **duplicate removals** on `s` until we no longer can.

Return the final string after all such duplicate removals have been made. It is guaranteed that the answer is unique.

Example 1:

```
Input: s = "abcd", k = 2

Output: "abcd"

Explanation: There's nothing to delete.
```

Example 2:

```
Input: s = "deeedbbcccbdaa", k = 3

Output: "aa"

Explanation:

First delete "eee" and "ccc", get "ddbbbdaa"

Then delete "bbb", get "dddaa"

Finally delete "ddd", get "aa"
```

Example 3:

```
Input: s = "pbbcggttciiippooaais", k = 2

Output: "ps"
```

```java
class Solution {
    public String removeDuplicates(String s, int k) {
        char[] arr = s.toCharArray();
        int j=0;
        int[] f = new int[s.length()];
        for(int i =0; i<s.length(); ++i, ++j){
            arr[j]=arr[i];
            f[j] = j>0 && arr[j-1] == arr[j] ? f[j-1]+1:1;
            if(f[j]==k){
                j-=k;
            }
        }
        return String.copyValueOf(arr, 0, j);
    }
}
```

2255. Count Prefixes of a Given String

You are given a string array `words` and a string `s`, where `words[i]` and `s` comprise only of **lowercase English letters**. Return *the **number of strings** in* `words` *that are a **prefix** of* `s`.
A **prefix** of a string is a substring that occurs at the beginning of the string. A **substring** is a contiguous sequence of characters within a string.

Example 1:

```
Input: words = ["a","b","c","ab","bc","abc"], s = "abc"

Output: 3

Explanation:

The strings in words which are a prefix of s = "abc" are:

"a", "ab", and "abc".
```

Example 2:

```
Input: words = ["a","a"], s = "aa"

Output: 2
```

```java
// 1st Approach
class Solution {
    public int countPrefixes(String[] words, String s) {
        int count=0;
        for(String str : words){
            if(s.indexOf(str)==0) count++;
        }
        return count;
    }
}
```

```java
// 2nd Approach
class Solution {
    public int countPrefixes(String[] words, String s) {
        int count=0;
        for(String str : words){
            if(isPrefix(s, str)) count++;
        }
        return count;
    }
    public static boolean isPrefix(String haystack, String needle){
        if(haystack.length()==0 || needle.length()==0){
            return false;
        }
        for(int j=0; ; j++){
            if(j==needle.length()) return true;
            if((j)==haystack.length()) return false;
            if(haystack.charAt(j)!=needle.charAt(j)) break;
        }
        return false;
    }
}
```

1946. Largest Number After Mutating Substring

You are given a string `num`, which represents a large integer. You are also given a **0-indexed** integer array `change` of length `10` that maps each digit `0-9` to another digit. More formally, digit `d` maps to digit `change[d]`.

You may **choose** to **mutate a single substring** of `num`. To mutate a substring, replace each digit `num[i]` with the digit it maps to in `change` (i.e. replace `num[i]` with `change[num[i]]`).

Return *a string representing the* ***largest*** *possible integer after* ***mutating*** *(or choosing not to) a* ***single substring*** *of* `num`.

A **substring** is a contiguous sequence of characters within the string.

Example 1:

```
Input: num = "132", change = [9,8,5,0,3,6,4,2,6,8]

Output: "832"

Explanation: Replace the substring "1":
- 1 maps to change[1] = 8.

Thus, "132" becomes "832".

"832" is the largest number that can be created, so return it.
```

Example 2:

```
Input: num = "021", change = [9,4,3,5,7,2,1,9,0,6]

Output: "934"

Explanation: Replace the substring "021":
```

```
class Solution {
    public String maximumNumber(String num, int[] change) {
        char[] chars = num.toCharArray();   // Copy value in char Array
        boolean changed = false;
        for(int i=0; i<chars.length; i++){
            // chars = 132 --> 1=49(ascii) & 0=48(ascii)--> 49-48=1
            int prev = chars[i]-'0';
            int list = change[prev];

            if(prev<list){
                chars[i]=(char)(list + '0');
                changed = true;
            }
            // we need contiguous, so if change have been done previously
            // and if currently no changing, so break the loop, return it
            if(prev>list && changed){
                break;
            }
        }
        return String.copyValueOf(chars);
    }
}
```

67. Add Binary

Given two binary strings `a` and `b`, return *their sum as a binary string*.

Example 1:

```
Input: a = "11", b = "1"

Output: "100"
```

Example 2:

```
Input: a = "1010", b = "1011"

Output: "10101"
```

```
class Solution {
    public String addBinary(String a, String b) {
        StringBuilder sb = new StringBuilder();
        int i=a.length()-1, j=b.length()-1;
        int carry=0;

        while(i>=0 || j>=0){
            int sum=carry;
            if(i>=0){
                sum+=a.charAt(i--)-'0';
            }
            if(j>=0){
                sum+=b.charAt(j--)-'0';
            }
            sb.append(sum%2);
            carry=sum/2;
        }
        if(carry!=0){
            sb.append(carry);
        }
        return sb.reverse().toString();
    }
}
```

387. First Unique Character in a String

Given a string `s`, *find the first non-repeating character in it and return its index*. If it does not exist, return `-1`.

Example 1:

```
Input: s = "leetcode"

Output: 0
```

Example 2:

```
Input: s = "loveleetcode"

Output: 2
```

```java
class Solution {
    public int firstUniqChar(String s) {
        int[] f = new int[26];
        for(int i=0; i<s.length(); i++){
            f[s.charAt(i)-'a']++;
        }
        for(int i=0; i<s.length(); i++){
            if(f[s.charAt(i)-'a']==1) return i;
        }
        return -1;
    }
}
```

```java
class Solution {
    public int firstUniqChar(String s) {
        Queue<Character> ch = new LinkedList<Character>();
        int len = s.length();
        char temper = 0;
        for(int j = 0; j < s.length(); j++)
        {
            ch.offer(s.charAt(j));
        }
        while(len > 0)
        {
            temper = ch.poll();
            if(!ch.contains(temper))
            {
                return s.indexOf(temper);
            }else{
                ch.offer(temper);
            }
            len--;
        }
        return -1;
    }
}
```

383. Ransom Note

Given two strings `ransomNote` and `magazine`, return `true` *if* `ransomNote` *can be constructed by using the letters from* `magazine` *and* `false` *otherwise*.

Each letter in `magazine` can only be used once in `ransomNote`.

Example 1:

```
Input: ransomNote = "a", magazine = "b"

Output: false
```

Example 3:

```
Input: ransomNote = "aa", magazine = "aab"

Output: true
```

```java
class Solution {
    public boolean canConstruct(String ransomNote, String magazine) {
        int[] f = new int[26];
        for(int i=0; i<magazine.length(); i++){
            f[magazine.charAt(i)-'a']++;
        }
        for(int i=0; i<ransomNote.length(); i++){
            if(f[ransomNote.charAt(i)-'a']<=0) return false;
            f[ransomNote.charAt(i)-'a']--;
        }
        return true;
    }
}
```

```java
class Solution {
    public boolean canConstruct(String ransomNote, String magazine) {
        HashMap<Character, Integer> map = new HashMap<>();
        for(int i = 0; i < magazine.length(); i++){
            char c = magazine.charAt(i);
            map.put(c, map.getOrDefault(c,0)+1);
        }
        for(int i = 0; i < ransomNote.length(); i++){
            char c = ransomNote.charAt(i);
            if(!map.containsKey(c) || map.get(c) <= 0){
                return false;
            }
            map.put(c, map.get(c)-1);
        }
        return true;
    }
}
```

242. Valid Anagram

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
```

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

```
class Solution {
    public boolean isAnagram(String s, String t) {
        if (s.length() != t.length()) {
            return false;
        }
        HashMap<Character, Integer> hashMap = new HashMap<Character, Integer>();
        for (int i = 0; i < s.length(); i++) {
            hashMap.put(s.charAt(i), hashMap.getOrDefault(s.charAt(i), 0)+1);
            hashMap.put(t.charAt(i), hashMap.getOrDefault(t.charAt(i), 0)-1);
        }
        for (Integer count : hashMap.values()) {
            if (count != 0) {
                return false;
            }
        }
        return true;
    }
}
```

1844. Replace All Digits with Characters

You are given a **0-indexed** string `s` that has lowercase English letters in its **even** indices and digits in its **odd** indices.

There is a function `shift(c, x)`, where `c` is a character and `x` is a digit, that returns the `x`th character after `c`.

- For example, `shift('a', 5) = 'f'` and `shift('x', 0) = 'x'`.

For every **odd** index `i`, you want to replace the digit `s[i]` with `shift(s[i-1], s[i])`.

Return `s` *after replacing all digits. It is* ***guaranteed*** *that* `shift(s[i-1], s[i])` *will never exceed* `'z'`.

Example 1:

```
Input: s = "a1c1e1"

Output: "abcdef"

Explanation: The digits are replaced as follows:

- s[1] -> shift('a',1) = 'b'

- s[3] -> shift('c',1) = 'd'

- s[5] -> shift('e',1) = 'f'
```

```java
class Solution {
    public String replaceDigits(String s) {
        StringBuilder sb = new StringBuilder();
        for(int i=0; i<s.length(); i++){
            if(i%2==0){
                sb.append(s.charAt(i));
            }
            else{
                int a = (s.charAt(i)-'0');
                a = a+(int)s.charAt(i-1);
                String c = new Character((char)(a)).toString();
                sb.append(c);
            }
        }
        return sb.toString();
    }
}
```