Striver’s SDE Sheet

String Part-II

Find the Index of the First Occurrence in a String

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.

Ans:

Just a two pointer method

Just a sliding window concept with fixed size window

Code:

```
class Solution {
public:
    int strStr(string haystack, string needle) {

        for (int i = 0; i < haystack.length(); i++) {
            if (haystack[i] == needle[0]) {
                int i1 = i + 1;
                int j = 1;
                while (haystack[i1] && needle[j]) {
                    if (haystack[i1] != needle[j]) break;
                    i1++;
                    j++;
                }
                if (j == needle.length()) return i;
            }
        }
```

```
        return -1;
    }
};
```

Or just stl function .substr()

Minimum Characters For Palindrome

Given a string STR of length N. The task is to return the count of minimum characters to be added at front to make the string a palindrome. For example, for the given string "deed", the string is already a palindrome, thus, minimum characters needed are 0. Similarly, for the given string "aabaaca", the minimum characters needed are 2 i.e. 'a' and 'c' which makes the string "acaabaaca" palindrome

Ans:

Make two pointer and a counter to check dislocates char

Code:

```
#include<bits/stdc++.h>
int minCharsforPalindrome(string str) {
	int n = str.size();
	int i = 0, j = n - 1, count = 0, tempJ = j;
	while(i < j) {
		if(str[i] == str[j])
			i++, j--;
		else {
			count++;
			i = 0, tempJ--;
			j = tempJ;
		}
	}
	return count;
}
```

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.

Ans:

Just a frequency question (check out notes)

```
class Solution {

public:

    bool isAnagram(string s, string t) {

           if (s.size() != t.size()) return false;

        vector v1(26, 0);

        vector v2(26, 0);

        for(int i = 0 ; i < s.size() ; i ++){

            v1[s[i] - 'a']++;

            v2[t[i] - 'a']++;

        }


        return (v1 == v2);

    }

};
```

count-and-say

The **count-and-say** sequence is a sequence of digit strings defined by the recursive formula:

- countAndSay(1) = "1"
- countAndSay(n) is the way you would "say" the digit string from countAndSay(n-1), which is then converted into a different digit string.

To determine how you "say" a digit string, split it into the **minimal** number of substrings such that each substring contains exactly **one** unique digit. Then for each substring, say the number of digits, then say the digit. Finally, concatenate every said digit.

For example, the saying and conversion for digit string "3322251":

Ans:

Question suggest some recursion thing

Code:

```
string m[31]={""};
class Solution {
public:
    string countAndSay(int n) {
        m[1]="1";
        string next,prev;
        int i,j,len;

           //generating the say(i)
        for(i=2;i<n+1;i++)
        {
            if(m[i]!="")
                continue;
            next="";
            prev=m[i-1];
            len=prev.size();
            int count=1;
```

```
                //reading out the say(i-1) i.i string prev
            for(j=1;j<len;j++)
            {
                if(prev[j-1]==prev[j])
                    count++;
                else
                {
                    next=next+to_string(count)+prev[j-1];
                    count=1;
                }

            }
            next=next+to_string(count)+prev[j-1];
            m[i]=next;
        }
        return m[n];
    }
};
```

Compare Version Numbers

Given two version numbers, version1 and version2, compare them.

Version numbers consist of **one or more revisions** joined by a dot '.'. Each revision consists of **digits** and may contain leading **zeros**. Every revision contains **at least one character**. Revisions are **0-indexed from left to right**, with the leftmost revision being revision 0, the next revision being revision 1, and so on. For example 2.5.33 and 0.1 are valid version numbers.

To compare version numbers, compare their revisions in **left-to-right order**. Revisions are compared using their **integer value ignoring any leading zeros**. This means that revisions 1 and 001 are considered **equal**. If a version number does not specify a revision at an index, then **treat the revision as 0**. For example, version 1.0 is less than version 1.1 because their revision 0s are the same, but their revision 1s are 0 and 1 respectively, and 0 < 1.

Return the following:

- If version1 < version2, return -1.
- If version1 > version2, return 1.
- Otherwise, return 0.

Ans:

Code:

```
class Solution {
public:
    int compareVersion(string version1, string version2) {
        int i=0, j=0;
        int vs1= version1.size(), vs2=version2.size();
        while (true){
            int v1=0, v2=0;
            while (i<vs1 && version1[i]!='.'){
                v1*=10;
                v1+= (version1[i]-'0');
                i++;
            }
            while (j<vs2 && version2[j]!='.'){
```

```
                v2*=10;
                v2+= (version2[j]-'0');
                j++;
            }
            if (v1<v2)return -1;
            if (v1>v2)return 1;
            if (i>=vs1 && j>=vs2)break; //return 0;
            i++; j++;
        }
        return 0;
    }
};
```