Problem Solving On Arrays and Objects

Created	@October 11, 2022 8:11 PM
Class	
Type	
Materials	
Reviewed	☐

Problem 1

Valid Anagram - LeetCode

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 =

https://leetcode.com/problems/valid-anagram/

Two strings are called `ANAGRAM` of each other, if we can form one string by rearranging the characters of the other string.

Example:

s1 = RACE , s2 = CARE → these two strings are anagram of each

s1 = KEEN, s2 = KNEE → these two strings are anagram of each other

Now we have to write a function that takes input of two strings, and checks whether they are ANAGRAM of each other or not ? Return TRUE if the two input strings are ANAGRAMS else return false.

Solution:

Brute Force:

The problem says that anagrams are permutations of each other. So how about we can try to generate all possible `n!` permutations of the first string, and then check whether any of the permutations are equal to the second string.

Ex: s1 = RAC, s2 = CAR

All permutations of s1 - > [RAC, RCA, CAR, CRA, ARC, ACR] → and here one of the permutations is CAR, so we can return true.

Time: O(n!)

Can we optimise it ?

Optimisation 1:

If two strings are anagram of each other, then if we write permutations of both of them then we will be having 1 common permutation among them.

RAC → [RAC, RCA, CAR, CRA, ARC, ACR]

CAR → [CAR, CRA, ARC, ACR, RAC, RCA]

Both of them have got same permutations. But we are mainly interested in a specific permutation i.e. the one we get after sorting them.

Here `ACR` is the permutation we should observe carefully. If we sort s1 and s2 character by character, then we get this permutation `ACR`

Example:

s1 = RAT, s2 = CAR

if we sort both of them s1 → ART, s2 → ACR, both of them are not equal to each other, hence they are not anagrams.

Time: `O(nlogn)` where n is the length of the strings.

```
var isAnagram = function(s, t) {
    s = s.split("").sort().join("");
    t = t.split("").sort().join("");
    return s == t;
};
```

Can we optimise it even further ?

Optimisation 2:

If two strings are permutation of each other, then both of them will be having absolute same amount of characters.

Ex: CARE, RACE

In both of the strings above, we have 1 occurrence of `C` , 1 occurrence of `A` , 1 occurrence of `E` , and 1 occurrence of `R` .

Ex: LISTEN, SILENT

In both of the strings above there is

Character	Frequency
L	1
I	1
S	1
T	1
E	1
N	1

Ex: KEEN, KNEE

Character	Frequency
K	1
E	2
N	1

If there is any deviation in the frequency and occurrence of the characters then the strings won’t be anagram.

So, can we try to prepare this frequency map ? And then check the occurrences ?

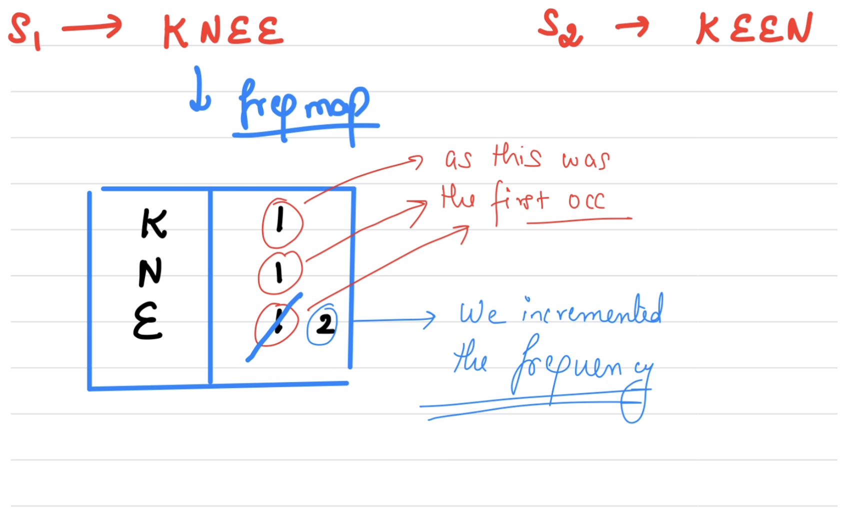
Here we can create a JS object, which will actually demonstrate the frequency map.

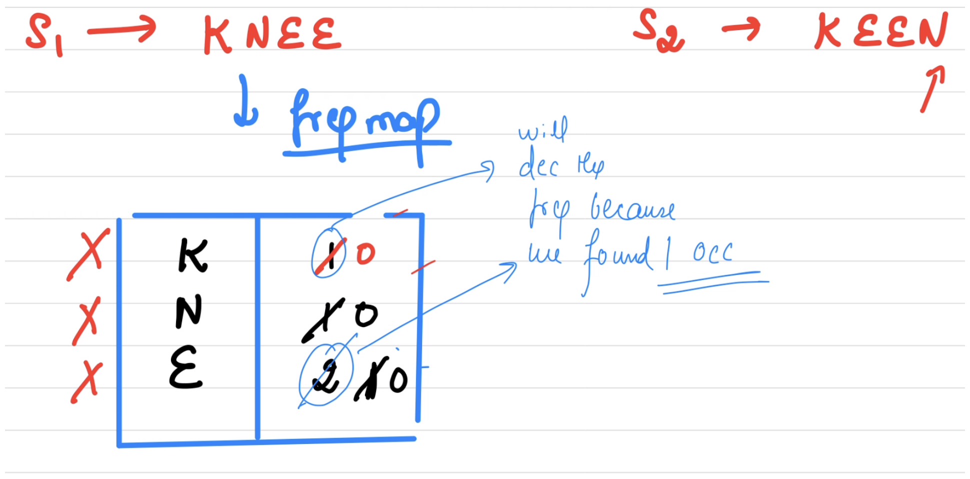
We can create frequency map of the first string.

Once we have the frequency map of first string, then we will iterate on the second string, and check whether it follows the frequency map or not ?

So to check the occurrences in string 2, we can one by one iterate on the string 2, and then we will try to check if the current character is present in frequency map or not. If not, then we will directly return a false, because we are sure strings are not anagram. If we found the character, we will decrease the frequency in map, because it signifies we found one successful occurrence.

Once the occurrence becomes 0, delete the entry. If two strings anagram, at last the size of frequency map should be 0.





```
/**
 * @param {string} s
 * @param {string} t
 * @return {boolean}
 */
var isAnagram = function(s, t) {
    if(s.length != t.length) return false;
    let freqMap = {};
    // lets fill the frequency map with s
    for(let i = 0 ; i < s.length; i++) {
        let currChar = s[i];
        if(!freqMap[currChar]) {
            // the current character is not present in the frequency map
            // we will create an entry with frequency 1
            freqMap[currChar] = 1;
        } else {
            // current character is already present in the map
            freqMap[currChar]++; // just increment the frequency
        }
    }
    // check the validity of t
    for(let i = 0; i < t.length; i++) {
        let currChar = t[i];
        if(!freqMap[currChar]) {
            // character is not even present, i.e. strings are not anagram
            return false;
        } else {
            // if the characters are present we will reduce the frequency
            freqMap[currChar]--;
            if(freqMap[currChar] == 0) {
                delete freqMap[currChar];
            }
        }
    }
    return Object.keys(freqMap).length == 0; // extract all the keys in an array and check
length
};
```

Time: `O(n)`

Space: `O(1)` // because maximum entries in the map will be equal to the number of unique characters in the language, independent of length of the string.

Problem 2:

Group Anagrams - LeetCode

Given an array of strings strs, group the anagrams together. You can return the answer in any order. An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase,

https://leetcode.com/problems/group-anagrams/

LeetCode

So having the knowledge of the Problem 1, we just solved we can some how drill down the solution for this problem.

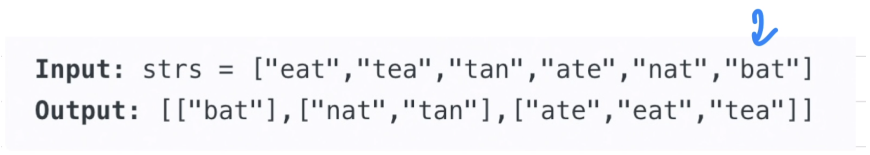
We will be having an array of strings, and we need to club the anagrams together.

If we want club the anagrams together, how about we try to think of something we is common in the anagrams, and based on this common thing we will try to map them.

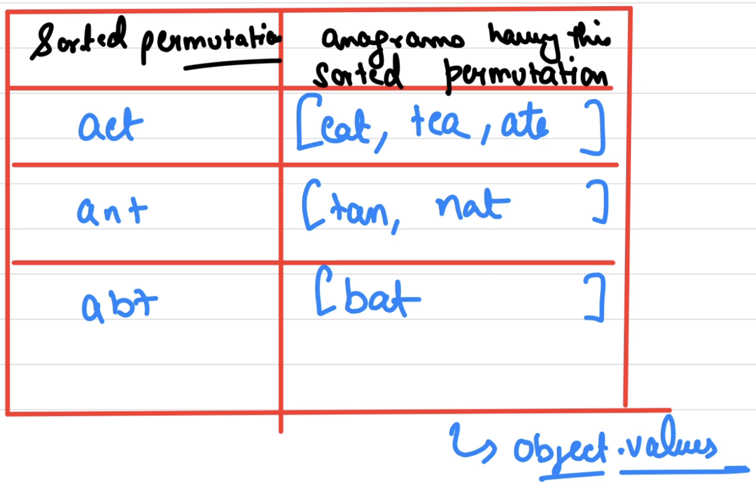
All the strings which are anagram of each other, if we sort them character by character, we will get same string.

So how about we create a mapping, where key can be the sorted string and value can be an array which will contain all the string which are anagrams and whose sorted arrangement will give us the key.

```
/**
 * @param {string[]} strs
 * @return {string[][]}
 */
var groupAnagrams = function(strs) {
    let mapping = {};
    for(let i = 0; i < strs.length; i++) {
        let curr = strs[i];
        let sortedStr = curr.split("").sort().join("");
        if(!mapping[sortedStr]) {
            // we donot have sorted permutation in the mapping
            // let's add it
            mapping[sortedStr] = [curr];
        } else {
            // we already have an entry of the sorted string in the map
            mapping[sortedStr].push(curr);
        }
    }
    return Object.values(mapping);
};
```



```
                                                       2
Input: strs = ["eat","tea","tan","ate","nat","bat"]
Output: [["bat"],["nat","tan"],["ate","eat","tea"]]
```



Sorted permutation	anagrams having this sorted permutation
act	[eat, tea, ate]
ant	[tan, nat]
abt	[bat]

↳ object.values

Time: O`(nklogk)` where n → length of array, k is the max length of any string

Space: `O(n)`

Can we optimise further ? Can we solve it in O(nk) ? In the previous problem, we did optimisation by avoiding sorting. Can we optimise here as well ?

Optimisation

We know that we are going to have limited unique characters per string i.e. 26 (given in the problem constraints).

If we write the frequency map of the string in the following form

```
a<freq>b<freq>c<freq>d<freq>e<freq>...........y<freq>z<freq> // frequencystring
```

Then the above string for two anagrams will be same.

So instead of using the sorted permutation as the key, can we use the above frequency string as the key and in the values we can store the original strings like before.

Example: [aet, tea, ate]

frequency string	array of anagrams

a1b0c0d0e1……..t1…..y0z0	[aet, tea, ate]

Only problem left is to generate the frequency string.

To generate this, we can first create a frequency map of the characters.

Then using that map, we can create the frequency string by iterating on the characters in sorted order.

This is how the mapping will look like

```
{
  a1b0c0d0e1f0g0h0i0j0k0l0m0n0o0p0q0r0s0t1u0v0w0x0y0z0: [ 'eat', 'tea', 'ate' ],
  a1b0c0d0e0f0g0h0i0j0k0l0m0n1o0p0q0r0s0t1u0v0w0x0y0z0: [ 'tan', 'nat' ],
  a1b1c0d0e0f0g0h0i0j0k0l0m0n0o0p0q0r0s0t1u0v0w0x0y0z0: [ 'bat' ]
}
```

```
/**
 * @param {string[]} strs
 * @return {string[][]}
 */
const chars = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'];
const createFreqString = function process(str) {
    let freqMap = {};
    for(let i = 0; i < str.length; i++) {
        const currChar = str[i];
        if(!freqMap[currChar]) {
            freqMap[currChar] = 1;
        } else {
            freqMap[currChar]++;
        }
    }
    let result = [];
    for(let i = 0; i < chars.length; i++) {
        // iterate for every engligh alphabet
        result.push(chars[i]); // first push the alphabet
        if(!freqMap[chars[i]]) {
            // if the alphabet is not present in map feed the frequency as 0
            result.push(0);
        } else {
            // otherwise feed the actual frequency
            result.push(freqMap[chars[i]]);
        }
    }
    return result.join("");
}
```

```
var groupAnagrams = function(strs) {
    let mapping = {};
    for(let i = 0; i < strs.length; i++) {
        let curr = strs[i];
        let frequencyString = createFreqString(curr); // change
        if(!mapping[frequencyString]) {
            // we donot have frequency string in the mapping
            // let's add it
            mapping[frequencyString] = [curr];// change
        } else {
            // we already have an entry of the frequency string in the map
            mapping[frequencyString].push(curr);// change
        }
    }
    return Object.values(mapping);
};
```

So we can just update the previous implementation, and instead of using a sorted string as key, we can try to generate frequency string.

But here we can raise a question (and quite an important one). We are first creating result array and then converting it to a string in the `createFreqString` function. Why not just directly create a string ?

Because in JS, strings are immutable. Due to this, whenever we do a concatenation, we create a new string in the memory by copying the content of the previous string and then adding the new concatenated part, this takes `O(n)` time where as adding in an array is `O(1)` , also due to the creation of new strings, we do a bit of space wastage.