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 ?

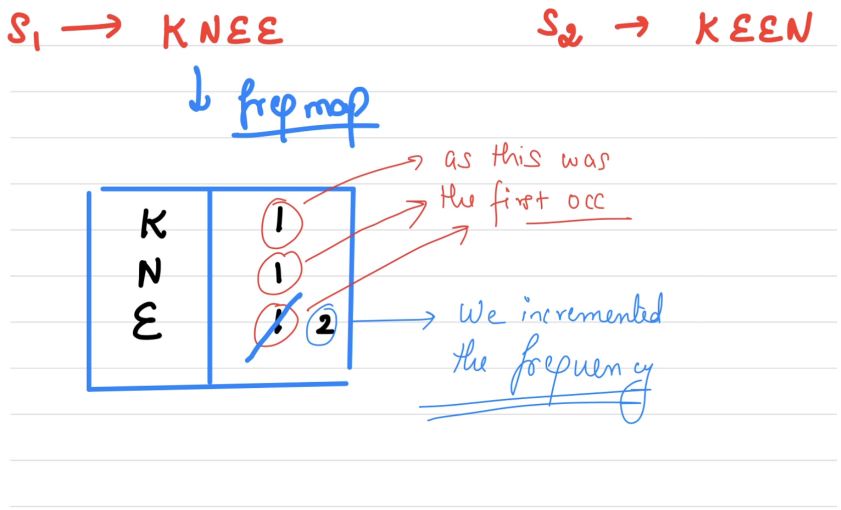
Here we can create a JS object, which will actually demonstrate the frequency map.

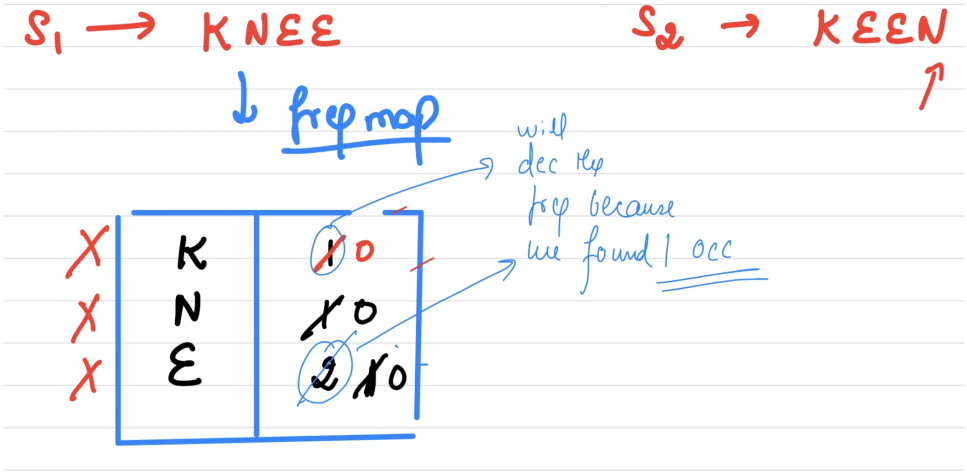
We can create frequency map of the first string.

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/

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.

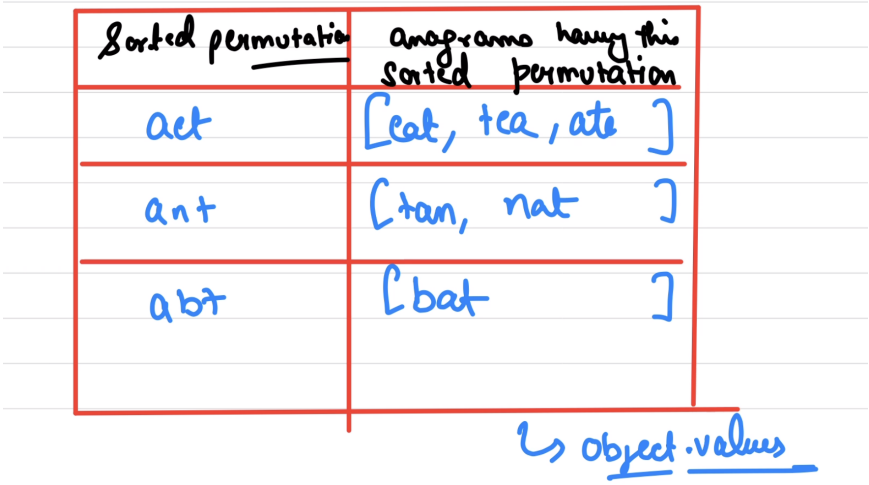
If we want club the anagrams together, how about we try to think of something we is common in the anagrams, and based on this common thing we will try to map them.

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);
};
```

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



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("");
}
```

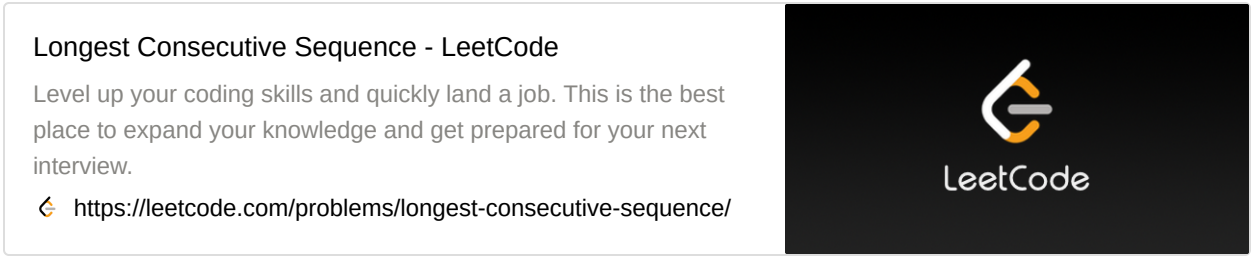
```
var groupAnagrams = function(strs) {
    let mapping = {};
    for(let i = 0; i < strs.length; i++) {
        let curr = strs[i];
        let frequencyString = createFreqString(curr); // change
        if(!mapping[frequencyString]) {
            // we donot have frequency string in the mapping
            // 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.

Problem 3



Given an unsorted array, find the length of the longest consecutive sequence in `O(n)` time.

Example: [19,17,1,3,18,2,5,6,16,15,14]

Ans: 6 i.e. [14,15,16,17,18,19]

Simple solution:

So because we can visualise a consecutive sequence as a sorted set of elements, we can somehow use sorting in order to solve the problem.

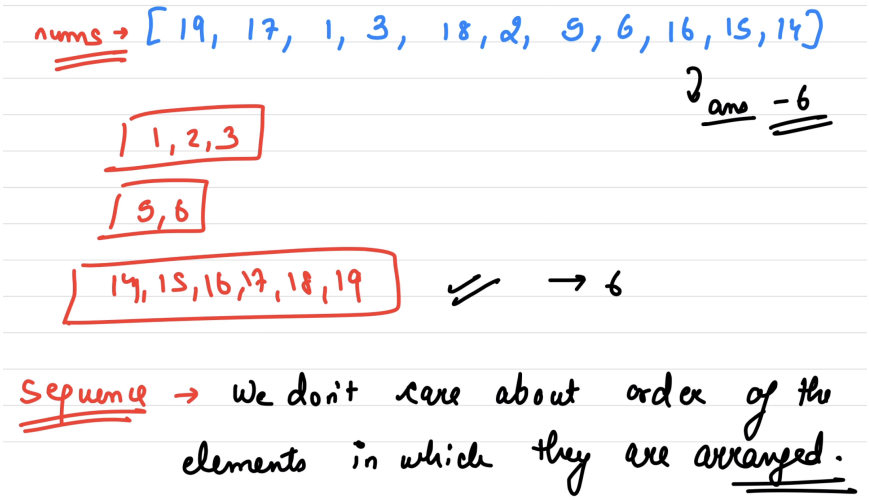
We can sort the array, and then we can iterate one by one starting from index 1, and check that

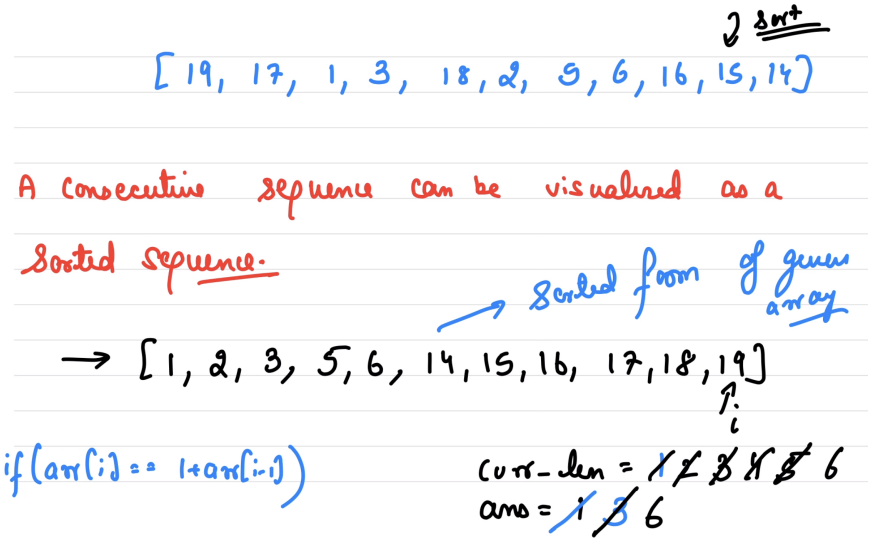
> `if (arr[i] == arr[i-1])` if this condition holds true then we are able to find one more element for the current sequence
>
> Once the condition is false, we can say that the previous consecutive sequence ends at ith index, and a new sequence starts from ith index.

One small observation will be that answer will be atleast 1 or greater than one for a non empty array.

So we can maintain two variables, `currLength` and `answer` that will help us to maintain current consecutive sequence length and overall answer.

Time: `O(nlogn)`



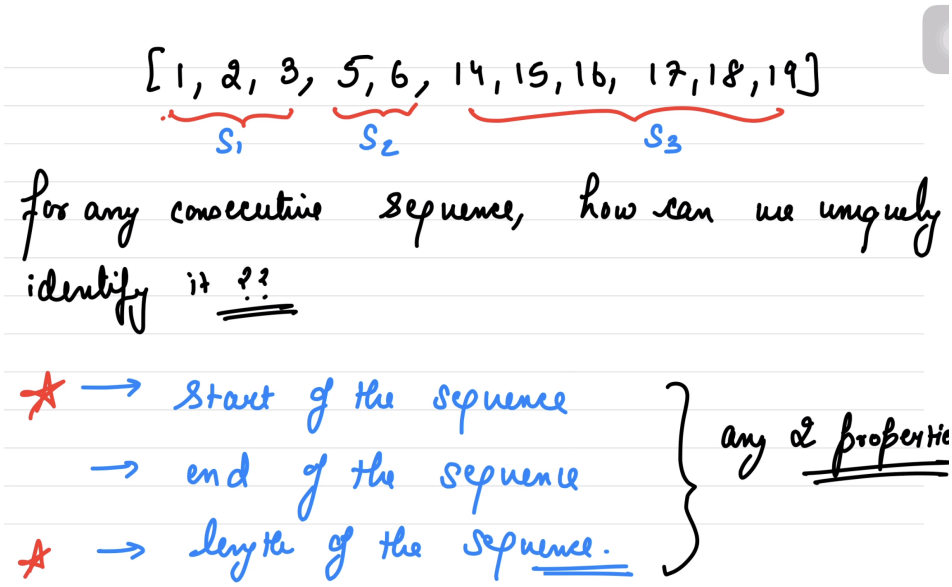


How to optimise ?

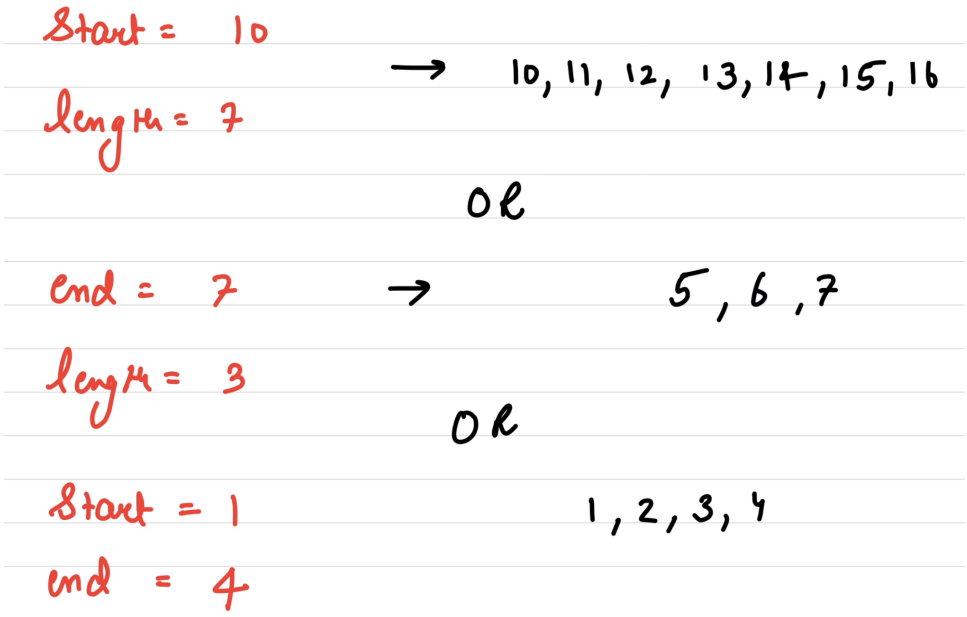
Optimisation:

We can observe that in order to identify any consecutive sequence uniquely we need to have any two from the following properties

- Start of the sequence
- End of the sequence
- Length of the sequence



We can use any two of the three properties to identify a sequence

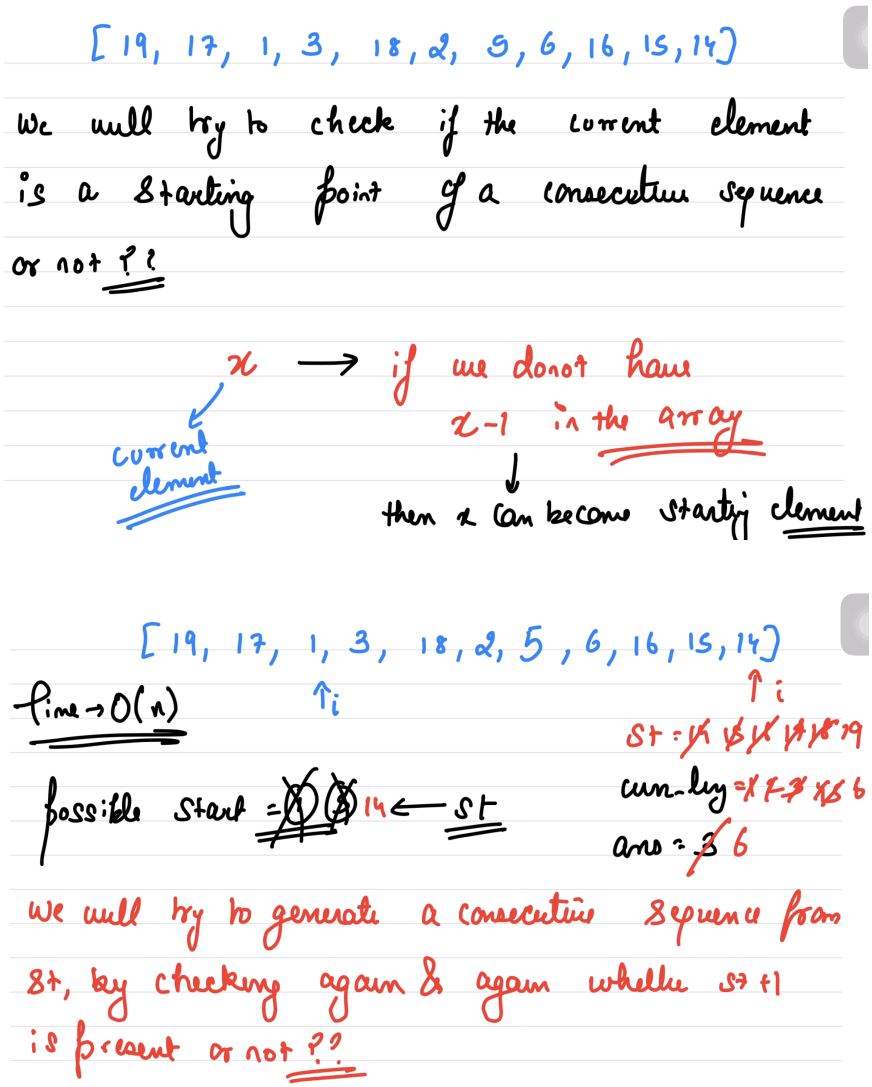


So can we somehow use this observation to solve our problem ?

We will one by one try to check whether current element of the array can become a potential starting point or not ?

How to check this ? If `curr-1` doesn't exist in the array then `curr` will be a potential starting point.

Once we found a potential start, we can try building the consecutive sequence out of it. We can one by one check if start+1 exists in the array or not ? If yes then we found one more element of the sequence. And we will update start and repeat this thing till the time we come to a position where element doesn't exist, and this point will signify that we have ended the current consecutive sequence, and we can compare the length of the sequence with the overall answer.



So once we get a potential start, we will put while loop to check whether start+1 exists or not ? If yes then we update start to start + 1 and increment length of the current sequence as well. Once we found that we no more got the element in the array, then length variable will denote the length of the consecutive sequence we were traversing.

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var longestConsecutive = function(nums) {
    if(nums.length == 0) return 0;
    let ans = 1; // ans will be atleast 1
    let mapping = {};
    for(let i = 0; i < nums.length; i++) {
        // create mapping of the elements
        let currElement = nums[i];
        mapping[currElement] = true;
    }
    for(let i = 0; i < nums.length; i++) {
        // loop will one by one check whether current element can be starting point or not
?
        let curr = nums[i];
        if(!mapping[curr - 1]) {
            // if curr - 1 doesn't exist in the array then current will be the starting po
int
            let start = curr;
            let len = 1;
            while(mapping[start+1]) {
                // till the time we have start+1 in the array, we will grow the sequence
                start++;
                len++;
            }
            // when loop ends, means we ended the sequence, so compare length
            ans = Math.max(ans, len);
        }
    }
    return ans;

};
```

What is the time complexity ?

it looks like we have a while inside a for so you might think it is a O(n^2) solution. But it is not. If we carefully analyse then we touch every element at max twice.

Once when we check whether the element can become potential start or not ?

The next time we touch it when we are building the sequence of which current element is part of .

So if there are n elements we have total 2n iteration i.e. Time: `O(n)`

Problem 4:

First Unique Character in a String - LeetCode

Level up your coding skills and quickly land a job. This is the best place to expand your knowledge and get prepared for your next interview.

https://leetcode.com/problems/first-unique-character-in-a-string/

Given a string find the index of the first non repeating character

Ex: `leetcode`

Ans: 0

Brute force:

So we need to find the first non repeating character, how about for every character we can check whether there is another occurrence of that char in the string or not. If there is another occurrence then this char won't be the answer otherwise, if we found a character that is non repeating in the remaining string, then we can say that we don't need to search further and this char will be the ans.

For example: `loveleetcode`

Once we found that `v` is non repeating we don't need to check the remaining string. `v` is out answer.

Now for every character, we are checking the remaining string, i.e. we are in the worst case going to compare it with n-1 other characters, so for n chars, there will n-1 comparisons.

Time: n*(n-1) → `O(n^2)`

Can we optimise ?

Optimisation:

We are concerned about repeating and non-repeating chars. Hence we somehow should deal with frequency of the characters because frequency can straight away tell us what characters are repeating what are non repeating.

We can first create a frequency map, because it can store the frequency of all chars.

We can iterate over the string and go on every character one by one, and check the frequency of the char. If we found a char with frequency 1, then we return the index of it, as we don't need to search for any other character, and here we are able to get the fact that char is non repeating by checking the frequency map in O(1) time.

```
/**
 * @param {string} s
 * @return {number}
 */
var firstUniqChar = function(s) {
    // time: O(n) space: O(1)
    let mp = {}; // frequency map to store the frequency
    for(let i = 0; i < s.length; i++) {
        if(!mp[s[i]]) {
            // check if s[i] is not present in the map, then create an entry with freq 1
            mp[s[i]] = 1;
        } else {
            // else if it is present then increment the frequency
            mp[s[i]]++;
        }
    }
    for(let i = 0; i < s.length; i++) {
        if(mp[s[i]] == 1) {
            // check if char is having freq 1, then this is the ans
            return i; // return the index of the character
        }
    }
    return -1; // no char was having freq 1
};
```

Time: `O(n)` Because we will iterate on every character once to create the frequency map and then once to search for non repeating

Space: `O(1)` because in the worst case doesn't matter how long the string is we will be only having 26 entries in the object