Advanced Problems on Arrays and Objects

Created	@July 1, 2022 8:12 PM
Class	
Type	
Materials	
Reviewed	☐

Problem 1

Given a string, print the frequency of each unique character in the string.

Ex: RELEVEL

```
{R - 1, E - 3, L - 2, V - 1}
```

The problem has a very simple solution. Because we have to keep a mapping of the character with it's frequency, we can use a JS object to store this mapping in the memory.

We can initialise a JS object and then one by one iterate on every character of the string.

If the character is already present in the mapping, we will increase it's frequency and if the character is not present we will make an entry in the map, with the frequency as 1.

```
function getFrequency(str) { // length of the input string is n
/**
  *Time: O(n)
  *Space: O(1)
*/
```

```
    let mp = {}; // this object will store the frequencies
    for(let i = 0; i < str.length; i++) {
        if(!mp[str[i]]) {
            // if the current char is not present in my mapping, we make an entry
            mp[str[i]] = 1;
        } else {
            // if the char is already present, increase the frequency
            mp[str[i]] += 1;
        }
    }
    return mp;
}

console.log(getFrequency("relevel"));
```

Why the space complexity is O(1) ?

In order to understand the space, we should ask ourselves a question that where we are using extra memory ? In the object `mp` we are using extra memory. But if the length of the given string is n, what can be the maximum number of entries in `mp` ?

The maximum number of entries will be dependent on the number of unique characters.

`aaaaaaaaaaaaaaaaaaaaaa` → `{a: 20}`

The space complexity will be defined by the number of unique characters. In the worst case the number of unique characters will be equal to the number of characters in our dictionary. So if the length of the string is 10^5 then also the number of entries in `mp` will be max 52. Even if the length is 10^7, the number of entries will be max 52. So, by increasing the length of the string, the space is not increasing i.e. space is constant w.r.t change in input size. That’s why space is `O(1)`.

Problem 2

Given a string of characters, find the first non-repeating character in the string.

Ex: `macbook` → `m`

`relevel` → `r`

`level` → `v`

`solutions` → `l`

Observations:

- When we say we want to find non repeating character, then we are concerned about the frequency of the characters.
- Because we are concerned bout frequency, it makes sense to have a frequency map.
- Once we have the frequency map, we can at least answer what all characters are non repeating, by simply looking at those characters whose frequency is `1`.

Based on the observations above, how can we find the `first non repeating` ?

Which character will be the first non repeating ? The one which has frequency as 1, and among all the characters who has got frequency 1, the index of this first non repeating character will be the least.

Once we have the frequency map with us, we can start reading the string from index 0, and one by one for each character that we encounter, we check if the frequency is 1 or not ? The first character during the looping process which gets the frequency 1, is our answer.

```
function firstNonRepeating(str) {
// Time: O(n)
// Space: O(1)
    let mp = {};
    for(let i = 0; i < str.length; i++) {
        if(!mp[str[i]]) {
            mp[str[i]] = 1;
        } else {
            mp[str[i]] += 1;
        }
    }
    for(let i = 0; i < str.length; i++) {
    //we are going to each character and then checking their freq
        if(mp[str[i]] == 1) {
            return str[i];
        }
    }
    return -1;
}
```

```
console.log(firstNonRepeating("aabbcc"));

/**
 * str = level
 * {l:2, e:2, v:1}
 * i = 0, str[i] -> l -> freq - 2
 * i = 1, str[i] -> e -> freq - 2
 * i = 2, str[i] -> v -> freq - 1
 */
```

Problem 3

A subarray is a contiguous cross section of a given array. Example: [1,2,3]

`[1] [1,2] [1,2,3] [2] [3] [2,3]` these are the subarray. [1,3] is not a subarray of the given array because it is not contiguous.

[1,5,2] → [1] [5][2][1,5][5,2][1,5,2]

Given an array of integers, check if there is any subarray with sum of it's elements as `0`.

Example:

`[1,2,-1,2,-3]` → Yes , Because we have a subarray `[2,-1,2,-3]` where sum is `2 + (-1) + 2 + (-3)` which evaluates to `0`

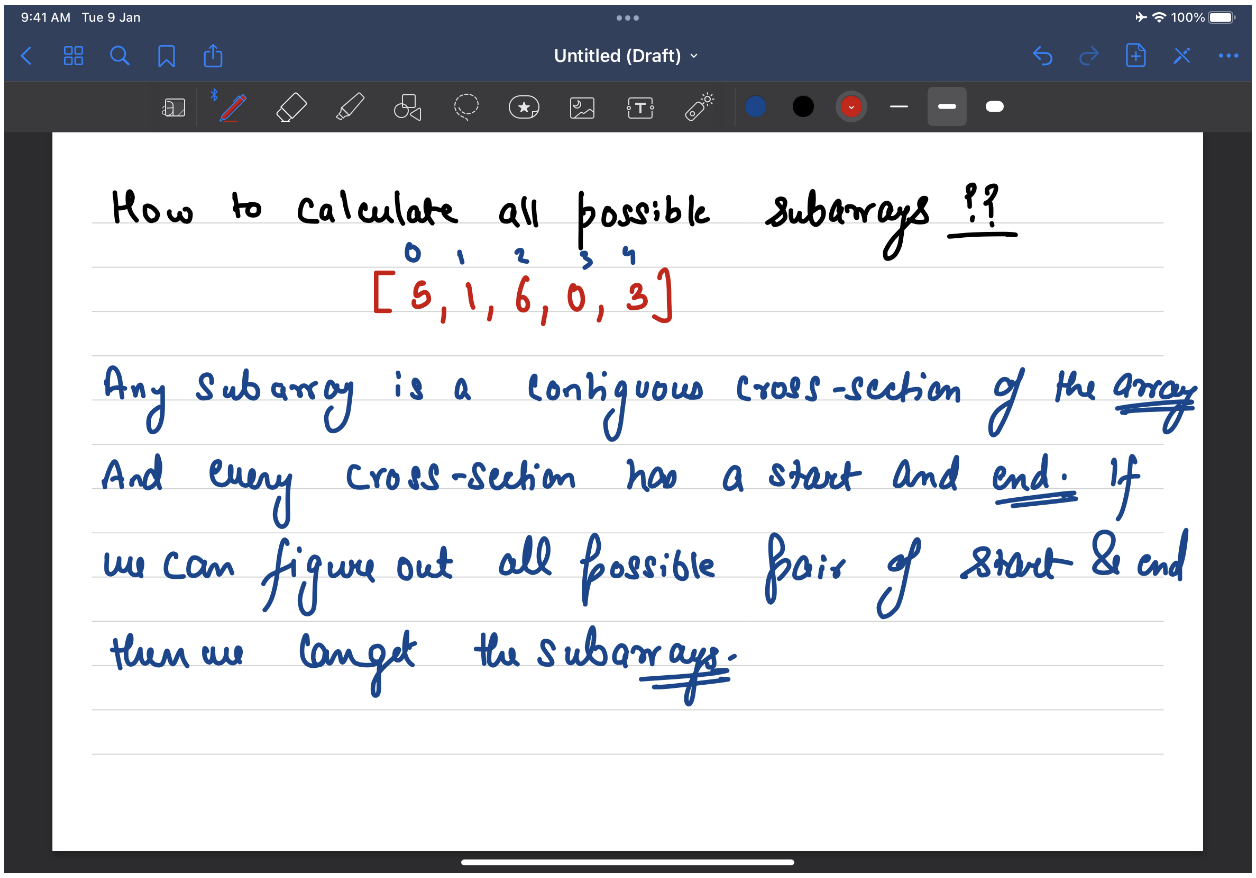
`[2,0,3,-3]` → yes, Because we have a subarray `[0]` which id having a sum zero, lso there is one more subarray `[3,-3]` whose sum is also 0

`[1,0,-1,5]` → yes, Because we have a subarray [1, 0, -1] which is having the sum 0

`[1,2,3,4,5]` → No, there is no subarray that sums up to 0

Brute force:

If we want to detect whether there is any subarray with sum 0, then how about if we can generate all the possible subarrays, and then take out their sum. The moment we find any subarray with sum 0, we return a true.



9:41 AM Tue 9 Jan

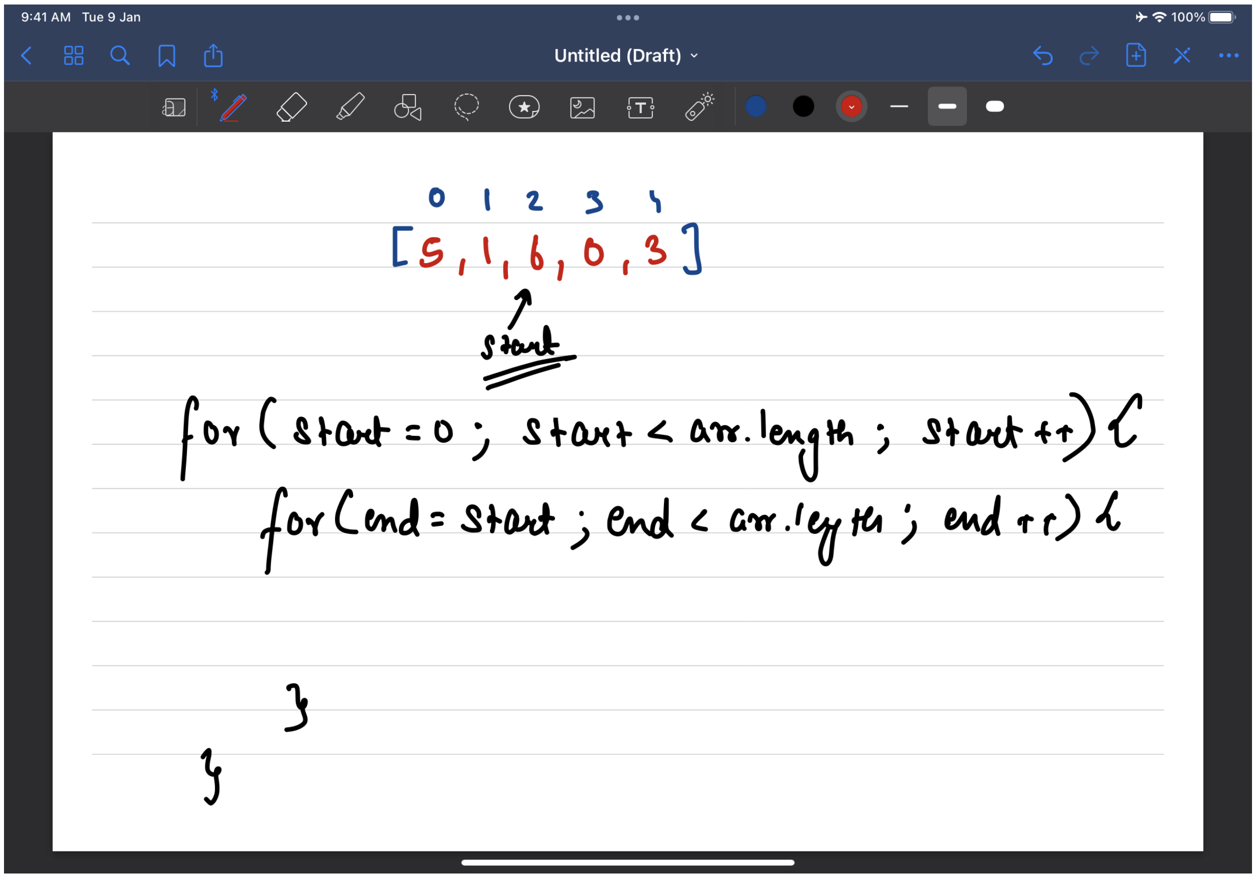
Untitled (Draft)

How to calculate all possible subarrays ??

$$\begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \\ 5, & 1, & 6, & 0, & 3 \end{array}$$

[5, 1, 6, 0, 3]

Any Subarray is a contiguous cross-section of the array. And every cross-section has a start and end. If we can figure out all possible pair of start & end then we can get the subarrays.



```
   0   1   2   3   4
[  5 , 1 , 6 , 0 , 3  ]
           ↑
         start

for (start = 0 ; start < arr.length ; start ++) {
    for (end = start ; end < arr.length ; end ++) {

    }
}
```

```
for(let start = 0; start < arr.length; start++) {
        for(let end = start; end < arr.length; end++) {
            let str = "";
            for(let k = start; k <= end; k++) {
                str += arr[k] + ",";
            }
            console.log(str);
        }
    }
```

Using the outer two loops of start and end, we are defining the corner points of the subarray, and then using the loop of K, we are actually iterating over 1 subarray.

So the above code can be used to generate all the subarrays.

`Time: O(n^3)`

Now coming back to the brute for for checking if there is any subarray with sum 0

```
function checkSubarrayWithSum0(arr) {
    /**
```

```
     * Time: O(n^3) Space: O(1)
     */
    // we try to generate all possible aubarrays
    for(let start = 0; start < arr.length; start++) {
        for(let end = start; end < arr.length; end++) {
            let sum = 0;
            for(let k = start; k <= end; k++) {
                sum += arr[k];
            }
            if(sum == 0) return true;
        }
    }
    return false;
}

console.log(checkSubarrayWithSum0([5,2,1,-3]));
```

We can have a small improvement

```
function checkSubarrayWithSum0(arr) {
    /**
     * Time: O(n^2) Space: O(1)
     */
    // we try to generate all possible aubarrays
    for(let start = 0; start < arr.length; start++) {
        let sum = 0;
        for(let end = start; end < arr.length; end++) {
            sum += arr[end];
            if(sum == 0) return true;
        }
    }
    return false;
}
```

But still n^2 is bad, can we improve a bit more ?

Q → If we are given an array, and somebody gives us two indexes i, j and asks, the sum of elements starting from index i to index j. How can we answer it efficiently ?

```
[1,2,3,4]
i = 1, j = 3
2 + 3 + 4 -> 9
i = 1, j = 1 -> 2
i = 0, j = 2 -> 6
```

One way is we can iterate from i to j and get the sum,

But what if instead of just one query of i, j we get multiple queries of i, j

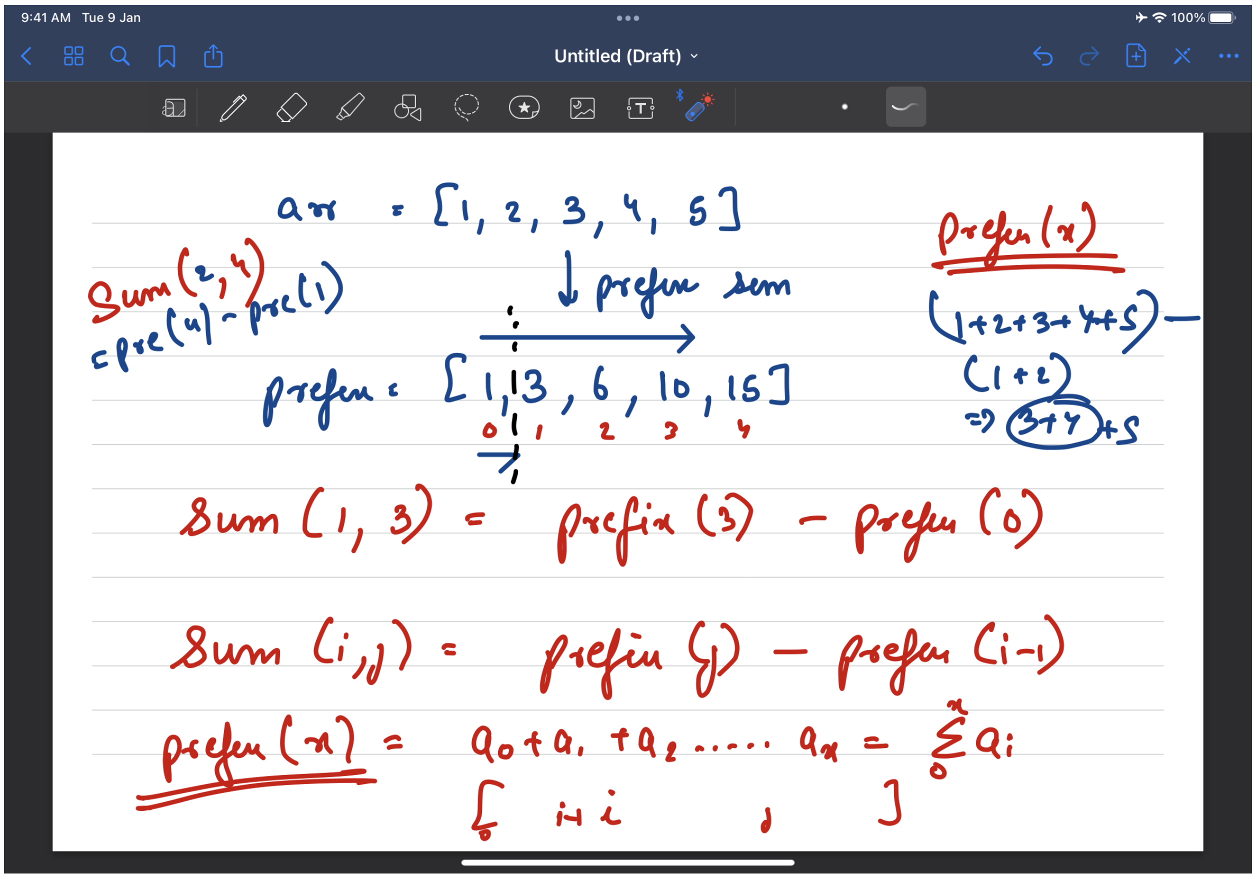
Total Q queries → O(QN) because for each query in the worst case we might have to iterate on the whole array.

Concept:

If we have to deal with sum of sub-arrays and we want to have sum of the values, then try to solve the problem using prefix sums.

```
[1,2,3,4,5] -> [1,3,6,10,15]
array -> prefix sum array
pre = [1,3,6,10,15]

sum(1, 3) => pre[3] - pre[0]
```


9:41 AM Tue 9 Jan
Untitled (Draft)
100%
arr = [1, 2, 3, 4, 5]
↓ prefix sum
prefix = [1, 3, 6, 10, 15]
0 1 2 3 4
Sum(2,4) = pre(4) - pre(1)
Prefix(x)
(1+2+3+4+5) —
(1+2)
⇒ (3+4)+5
Sum(1, 3) = prefix(3) − prefix(0)
Sum(i, j) = prefix(j) − prefix(i−1)
prefix(x) = a0 + a1 + a2 ..... ax = Σ ai
[i-1 i j]

In order to efficiently answer the sum of a subarray from index i, to j we can use the formulaT@PN

```
sum(i, j) = pre[j] - pre[i-1] // i-1 >= 0
sum(0, 5) = pre[5] // in this case where i is 0, just return pre[j]
```

Can we use this trick of prefix sum in the subarray question ?

Our agenda was to get a subarray, with sum 0

that means for some i, j

```
sum(i, j) = 0 // for any i, j -> we want to check if this holds true

sum(i, j) = pre[j] - pre[i-1]
```

```
pre[j] - pre[i-1] = 0

pre[j] = pre[i-1]
```

This means that if from our original array, we create a prefix sum array, and there repetition of the elements in the prefix sum array, then we have a subarray with sum 0.

Example: `[5,2,-3,1,6]` → `[5,7,4,5,11]` if i = 1, and j = 3 sum(1, 3) = 0 because

pre[1-1] = pre[3]

Corner case → `[0,1,2,3]` → `[0,1,3,6]`

If in the prefix sum array any element is repeated or any element is 0, then the answer is YES else no

`[3,4,-2,-4,2,5]` → `[3,7,5,1,3,8]`

pre[0] = pre[4] → i = 1, j = 4 → pre[i-1] = pre[j]

sum(i, j) → sum(1, 4) = 0 in the original array

code

for coding this solution there are two parts, first prepare the prefix sum array, second check the frequency of elements of prefix sum array.

`pre[i] = pre[i-1]+arr[i]`

[1,2,3,4,5] → [1,3,6,10,15]