Sliding Window - Notes

Sliding Window

(https://github.com/masai-codes/ds-algo/blob/master/notes/phase_3/3.1.1/3.1.1.b.md#prerequisites--complexity-analysis-brute-force) Prerequisites → Complexity Analysis, Brute Force

- ***Sliding window is an easy and intuitive optimisation technique, such that it maintains a window size, and that window size is such that is satisfies our requirement. The same window is then moved over through the entire data, providing the required answers***
- To understand sliding window, let's take the help of an example. Let us say that we want to find the individual sum of all subarrays of size K
- So, if the array given is `[1,2,3,4,5]` and the value of `K = 2`. So, all the subarrays of size 2, will be `1,2` , `2,3` , `3,4` , `4,5`. Therefore, the individual sum of all subarrays of size 2, will be `[3,5,7,9]`
- Now, the naive approach to solve this problem would be to generate all subarrays, and then filter out the subarrays of size 2. Finally, we find out the sum of all subarrays of the filtered subarrays
- The time complexity of the above solution will be $O(N^2)$. But, we are generating a lot of subarrays, which we don't really need for our solution
- This problem can be solved in linear time complexity with the help of sliding window technique
- Two pointer are maintained initially, one at index 0, and another at index `k-1` , and then moved through to the rest of the data, finding the sum of the K elements present in the window at a given time. The following code block shows the pseudo code for this approach

```
function slidingWindowSum(array,k){
   ans = []
         i  = 0
         j = k - 1
        while(j < array.length){
                        sum = 0
                        for (k = i;k<=j;k++) sum += array[k]
                        i++
                        j++
                        ans.add(sum)
        }
        return ans
}
```

- As you can see from the implementation, the time complexity of this approach is $O(N*K)$, and the space complexity is $O(1)$

(https://github.com/masai-codes/ds-algo/blob/master/notes/phase_3/3.1.1/3.1.1.b.md#subarray-

and-sum) Subarray and Sum

- In this problem, ***Given an array of integers of length n and a positive integer K, the task is to find the count of the longest possible subarrays with the sum of its elements not divisible by K***
- The intuitive approach to solve this problem would be to generate all subarrays of the given array, and find out the the longest subarrays having sum not divisible by K, and then how many subarrays exists of the given length
- But, the above approach is very costly in terms of the complexity analysis of the solution. This solution can be further improved. This problem can be broken down into the following subproblems
    - Finding the length of the longest subarray having sum not divisible by K
    - Finding the number of subarrays of the given length with sum not divisible by K
- The second problem can be solved in linear time using the sliding window, but for the first problem, we need to understand some facts
- A number if originally divisible by a number K, will remain divisible by K, if a number which is itself divisible by K, is subtracted from it. For example, the number 90 is divisible by 3. If we subtract 27 from it, which is divisible by 3, then the new value will also be divisible by 3. The value obtained from subtracting 27 from 90, is 63, which is divisible by 3
- Therefore, to make a sum not divisible by K, we have to subtract a number from K, which is not divisible by K. Therefore, in the array we try to find out the left most value not divisible by K, and also the right most value by K
- Our goal is to find the largest substring with sum not divisible by K, therefore, the sum remaining after removing the value before and including the left most value not divisible by K, will be not divisible by K. Similarly, the sum of the rest of the elements obtained after removing the right most element not divisible by K, and all the values after it, will also be not divisible by K
- For example, consider the array → [2,3,4,6], and K = 3. The left most element not divisible by K is 2, stored at index 0, while the right most element not divisible by K = 4, stored at index 2. So, we can either remove one element from the left, or two elements from right. But, since we want the subarray to be of maximum size, we will remove the element from the left
- Therefore, after removing one element from the array, then the size of the subarray, not divisible by K is 3, which is the largest possible, so we find the count of all subarrays which have size 3, and have a sum not divisible by K
- There are two edge cases to this approach, the first one being if the total sum of the array is not divisible by K, in that case, the longest subarray, with sum not divisible by K, is the array itself, and there can only be 1 subarray equal to the size of the array, therefore, the answer is 1
- The second edge case, is when the there are no elements not divisible by K, which makes it impossible to have a non empty subarray, with sum not divisible by K. Therefore, the count is 0 in that case. We need to ensure that these two edge cases are taken care of

```
function longestSubarray(array[],k){
          left = -1
                right = 0
                sum = 0;
                for (i = 0;i<array.length;i++){
                                if (array[i] % k != 0){
                                                if (left == -1) left = i
                                                right = i
                                }
                                sum += array[i]
                }
                if (sum % k != 0) return 1
                if (left == -1) return 0
                left = left + 1
                right = array.length - right
                max_window_size = array.length - min(left,right)
                return window_size
}
```

- After getting the maximum window size, the number of subarrays with maximum window size, and sum not divisible by K, can be found out by sliding window technique
- **Time Complexity** → $O(N)$, where N is the length of the array
- **Space Complexity** → $O(1)$