Sliding Window

Mustafa Muhammad

2nd October 2021

1. Maximum Subarray of Size K
2. First Negative Number in every Window of Size k
3. Count Occurences of Anagrams
4. Maximum of all subarrays of size k
5. Variable size Sliding Window — Largest Subarray of sum K
6. Longest Substring with k Unique Characters
7. Longest Substring without Repeating Characters
8. Pick Toys
9. Minimum Window Substring

Identification

1. Question on an array or string
2. Mentions subarray or substring
3. Fixed window size or condition

1 Max Sum SubArray of size K

```
class Solution:
    def maximumSumSubarray (self,K,Arr,N):
        max_sum = 0
        curr = 0
        for i in range(0, len(Arr) -K + 1):
            window = Arr[i:i+K]
            if i == 0:
                curr = sum(window)
            else:
                curr -= Arr[i-1]
                curr += Arr[i+K-1]


            max_sum = max(max_sum, curr)

        return max_sum
```

NOTE : Be careful with calculating the sum

2 First Negative Element in Every Window of Size K

```
def printFirstNegativeInteger( A, N, K):
    res = []
    temp = []
    j = 0
    i = 0
    while(j < len(A)):
        if A[j] < 0:
            temp.append(A[j])

        if j-i+1 < K:
            j+=1

        elif j-i+1 == K:
            if len(temp) != 0:
                res.append(temp[0])
            else:
                res.append(0)

            if A[i]<0:
                temp.pop(0)
            j+=1
            i+= 1

    return res
```

NOTE: BEST TO TRACE THIS SOLUTION OUT, CANT REALLY BE ATTEMPTED USING A FOR LOOP, SINCE THE LAST INCREMENT OF M IS TWICE.

3 Count Occurences of Anagrams

```
class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        res = []
        hash_map = {}
        hash_map_p = {}
        for c in p:
            if c in hash_map_p:
                hash_map_p[c] += 1
            else:
                hash_map_p[c] = 1
        i = 0
        j = 0
        while(j < len(s)):
            if s[j] not in hash_map:
                hash_map[s[j]] = 1
            else:
                hash_map[s[j]] += 1
            if j-i+1 < len(p):
                j+=1
            elif j-i+1 == len(p):
                if hash_map == hash_map_p:
                    res.append(i)

                hash_map[s[i]] -= 1
                if hash_map[s[i]] == 0:
                    del hash_map[s[i]]
                i+=1
                j+=1
        return res
```

Brute forcing is to create a window and sort, optimal way is to store previous input using a hashmap.

4 Maximum of all subarrays of size k

We make use a queue in order to reduce time complexity as brute force is too slow to pass all the test cases.

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        res = []
        i = 0
        j = 0
        ans = []
        queue = []
        if k > len(nums):
            return max(nums)

        while(j < len(nums)):
            # General removal to make room for new better numbers
            while(len(queue) > 0 and queue[-1] < nums[j]):
                queue.pop()
            queue.append(nums[j])
            if j-i+1 < k:
                j+= 1
            elif j-i+1 == k:
                res.append(queue[0])
                # Step to remove the current max e.g 3 or 5 and make way for new
                if nums[i] == queue[0]:
                    queue.pop(0)
                j += 1
                i += 1
        return res
```

You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3

[1 3 -1] -3 5 3 6 7 3

1 [3 -1 -3] 5 3 6 7 3

1 3 [-1 -3 5] 3 6 7 5

1 3 -1 [-3 5 3] 6 7 5

1 3 -1 -3 [5 3 6] 7 6

1 3 -1 -3 5 [3 6 7] 7

Output = [3, 3, 5, 5, 6, 6]

5 Variable Sliding Window

```
class Solution:
    def lenOfLongSubarr (self, nums, N, k) :
        prevs = {0:-1}
        sm= 0
        ans = 0
        for i in range(len(nums)):
            sm+=nums[i]
            if(sm-k in prevs):
                ans=max(i-prevs[sm-k],ans)
            if(sm not in prevs):
                prevs[sm]=i
        return ans
```

We store j in the hashmap, using the two pointer approach with i and j doesnt seem to work with negative values.

6 Gen Format - Fixed Sliding Window

```
def fixed_sliding_window():
    i = 0
    j = 0

    while(j < size):
        if minsize < j:
            j+= 1

        elif minsize == j:
            ans <- calculation

            ans -= arr[i]

            #Slide the window
            i += 1
            j += 1
    return ans
```

7 Gen Format - Variable Sliding Window

```
def fixed_sliding_window():
    i = 0
    j = 0

    while(j < size):
        if condition < j:
            j+= 1

        elif condition == j:
            ans <- calculation

            j += 1

        elif condition > k:
            while condition > k:
                ans -= arr[i]
                i += 1

            j += 1

    return ans
```

8 Longest Substring with K unique char

```
class Solution:
    def longestKSubstr(self, s, k):
        i = 0
        j = 0
        ans = 0
        unique = {}
        while(j < len(s)):
            if s[j] not in unique:
                unique[s[j]] = 1
            else:
                unique[s[j]] += 1
            if len(unique) < k:
                j+=1
            elif len(unique) == k:
                ans = max(j - i + 1, ans)
                j += 1
            elif len(unique) > k:
                while(len(unique) > k):
                    unique[s[i]] -= 1
                    if unique[s[i]] == 0:
                        del unique[s[i]]
                    i += 1
                j += 1
        return -1 if ans == 0 else ans
```

Hash Set doesnt work because the question allows for multiple occurences.

9 Longest Substring without repeating Char

Using a Set.

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        if len(s) == 0:
            return 0
        unique = set()
        i =0
        j =0
        max_len = 0
        while(j < len(s)):
            if s[j] not in unique:
                unique.add(s[j])
                max_len = max(max_len, len(unique))
                j += 1

            elif s[j] in unique:
                unique.remove(s[i])
                i+=1
        return max_len
```

Using a HashMap.

```
    def lengthOfLongestSubstring(self, s: str) -> int:
        if len(s) == 0:
            return 0
        hash_map = {}
        i ,j, max_len = 0, 0, 0
        while(j < len(s)):
            if s[j] in hash_map:
                hash_map[s[j]] += 1
            else:
                hash_map[s[j]] = 1
            if len(hash_map) > j-i+1:
                j+=1
            elif len(hash_map) == j-i+1:
                max_len = max(max_len, j-i+1)
                j+=1
            elif len(hash_map) < j-i+1:
                while len(hash_map) < j-i+1:
                    hash_map[s[i]]-=1
                    if hash_map[s[i]] == 0:
                        del hash_map[s[i]]
                    i+=1
                j+=1
        return max_len
```

10 Minimum Sliding Window

```
class Solution:
    def minWindow(self, s: str, t: str) -> str:
        target_map = {}
        for c in t:
            if c in target_map:
                target_map[c] += 1
            else:
                target_map[c] = 1
        i = 0
        j = 0
        x = -1
        y = -1
        min_len = float('inf')
        count = len(target_map)
        while j < len(s):
            if count > 0:
                if s[j] in target_map:
                    target_map[s[j]] -=1
                    if target_map[s[j]] == 0:
                        count -= 1
            j+=1
            if count == 0:
                while count == 0:
                    if j-i+1 < min_len:
                        min_len = j-i+1
                        x = i
                        y = j
                    if s[i] in target_map:
                        target_map[s[i]] += 1
                        if target_map[s[i]] > 0:
                            count += 1
                    i+=1
        return s[x: y]
```

We use the count varible to see when we hit the window condition. And then proceed to decrement the i counter.