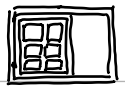
# Sliding Window 

## ○ Count Occurances of Anagrams :

update j
update i if needed

```
while ( j < n )
    mp[ s[j] ] - - ; if (s[j] in anagram)
    if ( j - i + 1 > lenAnagram)
        mp[s[i]] - - ; if (s[i] in anagram)
        i++ ;
```

update ans
j++

```
    if ( j - i + 1 == lenAnagram & mp all 0's ) ans++ .
j++
```

& Instead of this use counter.

## ○ Find all Anagrams (return indices)

## ○ Minimum subarray Sum >= Target :

```
while (j < n)
    s += nums[j]
    while ( s >= Target )
        ans = min(ans, j - i + 1)
        s -= s[i]
        i++ ;
j++
```

## ○ Find First -ve element in window size K :

```
while ( j < n )
    if ( nums[j] < 0 ) v . push back (j).
    if ( j - i + 1 == K )
        if ( v . size() <= p) ans . push b (0)
        else ans . push b ( v[p] )
        if ( nums[i] < 0 ) p++ ;
        i++
j++ ;
```

instead of p can use Queue & remove front.

## ○ Minimum Window Substring — Hard

```
while (     )
{   // update j th counter
    while (counter == 0 )
    {   if ( j - i + 1 < ans Min) .....
        // update i th index // }
j++
}
```

# Contains Duplicate II

# Count Subarrays with fixed Bounds  Hard

nums: 6 3 5 2 1  minK = 1  maxK = 5

ans = 2

subarray having min = minK
max = maxK.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 7 | 2 | 5 | 3 | 1 | 4 | 3 | 9 |
| culIdx | -1 | -1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 8 |
| minIdx | -1 | 0 | 0 | 0 | 0 | 0 | 5 | 5 | 5 | 5 |
| maxIdx | -1 | -1 | -1 | -1 | 3 | 3 | 3 | 3 | 3 | 3 |
| temp | | -1-(-1) = 0 | -1-1 = -2 | -2 | 0-1 = -1 | -1 | 3-1 = 2 | 3-1 = 2 | 3-1 = 2 | 3-8 = -5 |

[min(minIdx, MaxIdx) - culIdx]

-ve means first valid boundary: 1 is behind culprit: 7

means from Last culprit to first valid sub array

2 possible sub arrays are there

2 5 3 1 and

5 3 1

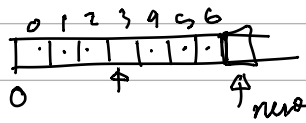
→ 2 5 3 1 4

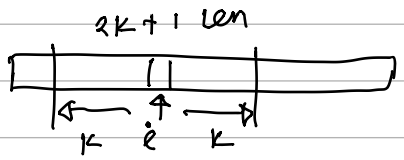
2+2 → 5 3 1 4

4 more → 2 5 3 1 4 3

valid subarrays possible. → 5 3 1 4 3

# K radius Subarray Averages




```
if (j >= 2*K)
    ans[j-K] = sumTill / (2K+1)
    sum -= nums[i];
    i++;
```

# Maximum 1's after Deleting 1-zero :-

```
while (cnt > 1)
{   if(nums[i]==0) cnt--
    i++
}
ans = max(ans, j-i)
```

2nd approach

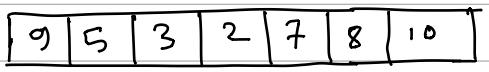
```
lastSeen = -1
if(nums[j] == 0){
    if (lastSeen != -1)
        i = lastSeen + 1
        lastSeen = j
    else lastSeen = j }
ans = max(j-i)
```

## Maximize Confusion in Exam :-

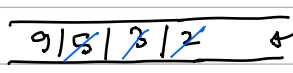
"T T F T F T F F F" maximum K no. of Flips are possible T→F, F→T

```
{ while cntT, cntF (.....)++
  while ( min(cntT, cntF) > K)
        cntT (....)--
        cntF (....)--
        i++
  ans = max (ans, j-i+1)
  j++ }
```

## Sliding Window Maximum ( Monotonic Deque ) Hard




processing 7




deque

9 7

when i moves
it will remove front of deque

## Frequency of the most Frequent Element :-

arr : [ 2, 5, 10, 13, 17 ] operations = 5.

arr[i] can be changed from x → x+K using K operation points
return freq of max element

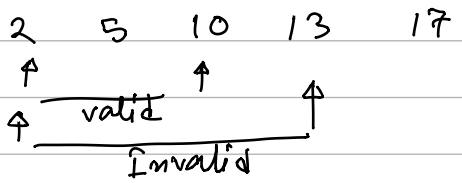
→ since we can't decrement ; max freq must be of some bigger no.

[ 2 5 10 13 17 ]

binary search here to get window len

try 13

or, use sliding window for obtaining max window.




```
while (j < n)
    sm += nums[j]
    if (while) ( (j-i+1) * nums[j] - sm > K)
        sm -= nums[i]
        i++;
    ans = max(ans, j-i+1)
```

instead of while; if can also work.

## ○ Count Complete Substrings :-

"igigee" — count substring where no 2 consecutive characters are more different than 2

in an subarray all character must be present exactly K no. of time.

① seperate → "aa|ddee|xyxy" — solve seperately

count no. of subarray where all element occur exactly K no. of time

② # need to write the sliding window code seperately.

solver(str, start, end, K) → return this

method; count all subarray having 1 unique character

" " " " 2 " "

⋮

min(26, no. of unique char available)

take fixed window length = K×1 for 1 unique chars

K×2 " 2 " "

K×3 " 3 " "

⋮

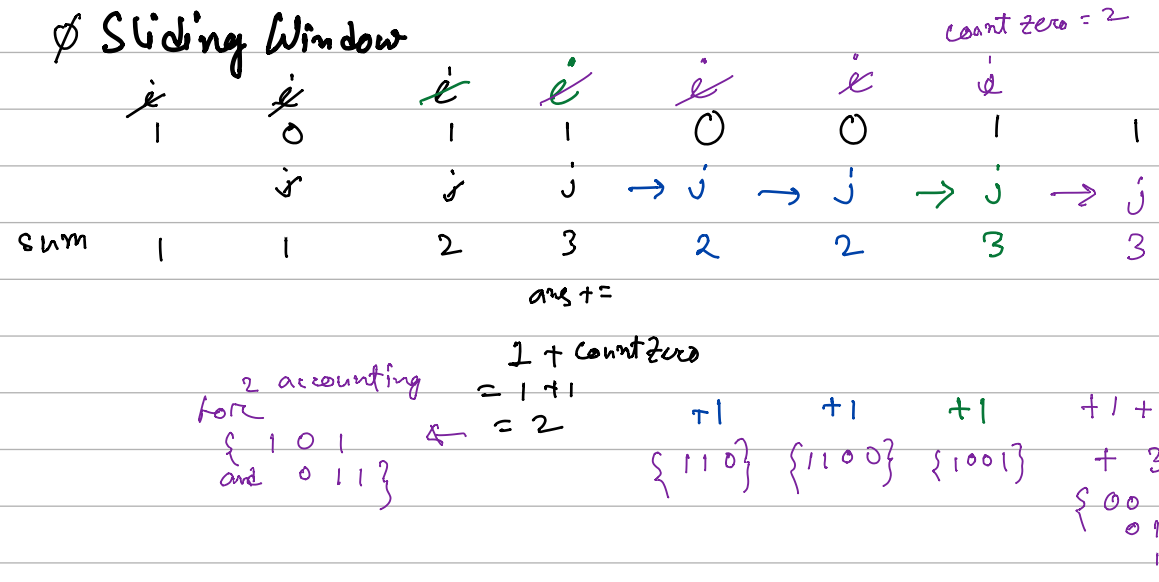
## ○ Binary Subarrays with Sum :-

nums [1, 0, 1, 1, 0, 0, 1, 1] target : 2

◊ PrefixSum (CumSum) + ~~Binary Search~~ Hash Table

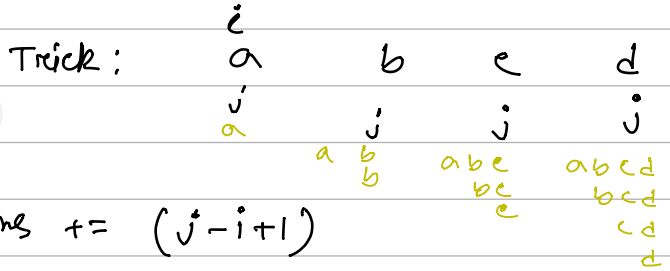
| | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|
| cumSum | 1 | 1 | 2 | 3 | 3 | 3 | 4 | 5 |
| ans | 0 | 0 | CumSum − Target (+1) | +2 | +2 | +2 | +1 | +3 |
| H.T. 0:1 | 0:1 | 0:1 | 0:1 (101) | 0:1 | 0:1 | 0:1 | 0:1 | 0:1 |
| | 1:1 | 1:2 | 1:2 | 1:2 [011] | 1:2 [0110] | 1:2 [01100] | 1:2 | 1:2 |
| | | | 2:1 | 2:1 | 2:1 | 2:1 | 2:1 [1001] | 2:1 |
| | | | | 3:1 | 3:2 | 3:3 | 3:3 | 3:3 [00111 01110 11100] |

## ∅ Sliding Window




## ○ Subarray product less than K

To count all subarrays a, b, c, d, ab, bc ... abcd

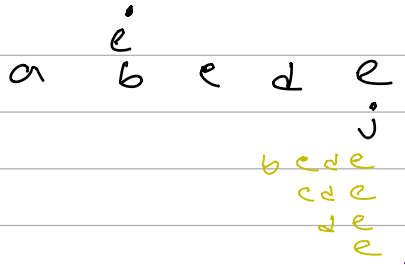



#⭐, #Revision

j : i → n ans += (j - i + 1)

1 + 2 + 3 + 4

next when i moves




## ❯ Similar Type

- → Maximum size subarray sum equals K
- → Subarray sum equals K.
- → Count subarrays having equal vowel & Consonents.
- → Longest Substring containing vowels in even Count.
- → Maximum no. of vowels in a substring of given length.

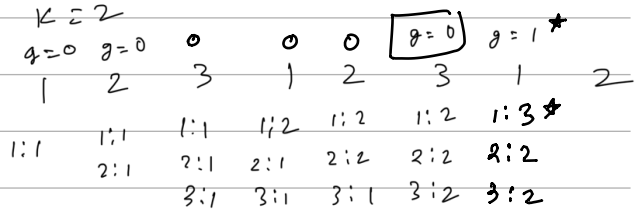
## 0 Length of longest Subarray with at most K frequency:

✓ good variable

✓ unordered_map

```
while → if (mp[nums[j]] == k)
            good --;
        mp[nums[j]] ++j;
        while( good != 0 )
            if (mp[nums[i]] == k+1)
                good ++j
            mp[nums[i]] --;
            i++
        ans = max(ans, j-i+1)
```

if can also work here since j-i+1 is not changing after achieving maximum




## 0 Count Subarrays where Element Appears atleast K times.

↳ element is map element here.

nums: [ 1, 3, 2, 3, 3 ] K = 2

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| | 1 | 3 | 2 | 3, j → | 3 |
| mapCount | 0 | 1 | 1 | 2 | j 2 |
| | i | i | i → | i | |
| ans | +(5-3)=2 | +(5-3)=2 | +(5-4)=1 | +(5-4)=1 | |
| accounting for | {1 3 2 3, 1 3 2 3 3} | {3 2 3, 3 2 3 3} | {2 3 3} | {3 3} | |

```
while ( j < n )
    elementCount += 1:0 if nums[j] = element.
    while ( elementCount == k )
        ans += n-j        → j to n-1 = n-1-j+1 = n-j
        elementCount -= 1:0 if nums[i] = elemt
        i++
    j++
```

### » Approach 2 using Hash table

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| | 1 | 3 | 2 | 3 | 3 | 2 |
| mp count | 0 | 1 | 1 | 2 | 3 | 3 |
| Table | | 1:1 | | 1:1 | 1:1 | 1:1 |
| | | | | 2:3 | 2:3 | 2:3 |
| | | | | | 3:3 | 3:3 |
| ans: | +0 | +1 | +0 | +2 | +4 | +4 |
| Accounting for | | | | {1 3 2 3 3 2 3} | {1 3 2 3 3 3 2 3 3} | {1 3 2 3 3 2 3 3 2 3 3 2} |

idx 0 to 1 (+2)

idx 0 to 3 (+4)

idx 0 to 3.

code :
```
for (int i=0 ; i<n i++ )
    if (nums[i] == element){
        elementCount ++ ;
        mp [elementCount] = i }
if ( elementCount >= K ) ans += mp [elementCount - K+1] +1
```

## 0 Subarrays with K Different Integers :- Hard

Good subarray consisting exactly K unique Integers
[1 2 3 1 1 2 3] → consist 3 integers.

# It's diffict to find exactly K element.
i-i+1 gives all subarrays ending at j (In these subarrays there will be all subarrays where at max K unique elements possible
i.e. subarrays having exactly K-1 unique elements, exactly K-2 unique elements ... etc.

$$ans = f(K) - f(K-1)$$

⇒ at max K = 2 elements,

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| | 1 | 2 | 1 | 2 | 3 | 4 |
| Table : | 1:1 | 1:1<br>2:1 | 1:2<br>2:1 | 1:2<br>2:2 | 1:~~2~~ 1 0<br>2:~~2~~ 1 | ⇒ 2:1<br>3:1 |
| | i | i | i | i | i | 2:~~1~~ 0<br>3:1<br>4:1 |
| ans : | +1 | +2 | +3 | +4 | +(j-i+1) = 2 | +2 |
| j-i+1 accounting for | {1}• | {1,2}•<br>{2}• | {1 2 1}•<br>{2 1}•<br>{1}• | {1 2 1 2}•<br>{2 1 2}•<br>{1 2}•<br>{2}• | {2 3}•<br>{3}• | {3 4}•<br>{4}• |

⇒ at max K = 3 elements,

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| | 1 | 2 | 1 | 2 | 3 | 4 |
| Table : | 1:1 | 1:1<br>2:1 | 1:2<br>2:1 | 1:2<br>2:2 | 1:2<br>2:2<br>3:1 | 1:~~2~~ 1 0<br>2:~~2~~ 1<br>3:1<br>4:1 |
| | i | i | i | i | i | |
| ans : | +1 | +2 | +3 | +4 | +5 | +(5-3+1) = 3 |
| j-i+1 | ~~{1}~~ | ~~{1 2}~~<br>~~{2}~~ | ~~{1 2 1}~~<br>~~{2 1}~~<br>~~{1}~~ | ~~{1 2 1 2}~~<br>~~{2 1 2}~~<br>~~{1 2}~~<br>~~{2}~~ | {1 2 1 2 3} ✓<br>{2 1 2 3} ✓<br>{1 2 3} ✓<br>~~{2 3}~~<br>~~{3}~~ | {2 3 4} ✓<br>~~{3 4}~~<br>~~{4}~~ |

exactly 3 elements : {1 2 1 2 3}, {2 1 2 3}, {1 2 3}, {2 3 4}

* using countOfDistinct instead of mp.size() is faster;

```
while( j<n )
    mp[nums[j]] ++ ;
    while( mp.size() > K )
        mp[nums[i]] -- ;
        if(mp[nums[i]]==0) mp.erase(nums[i])
        i++
    result += j-i+1
    j++
```

Approach 2 one pass : K=2

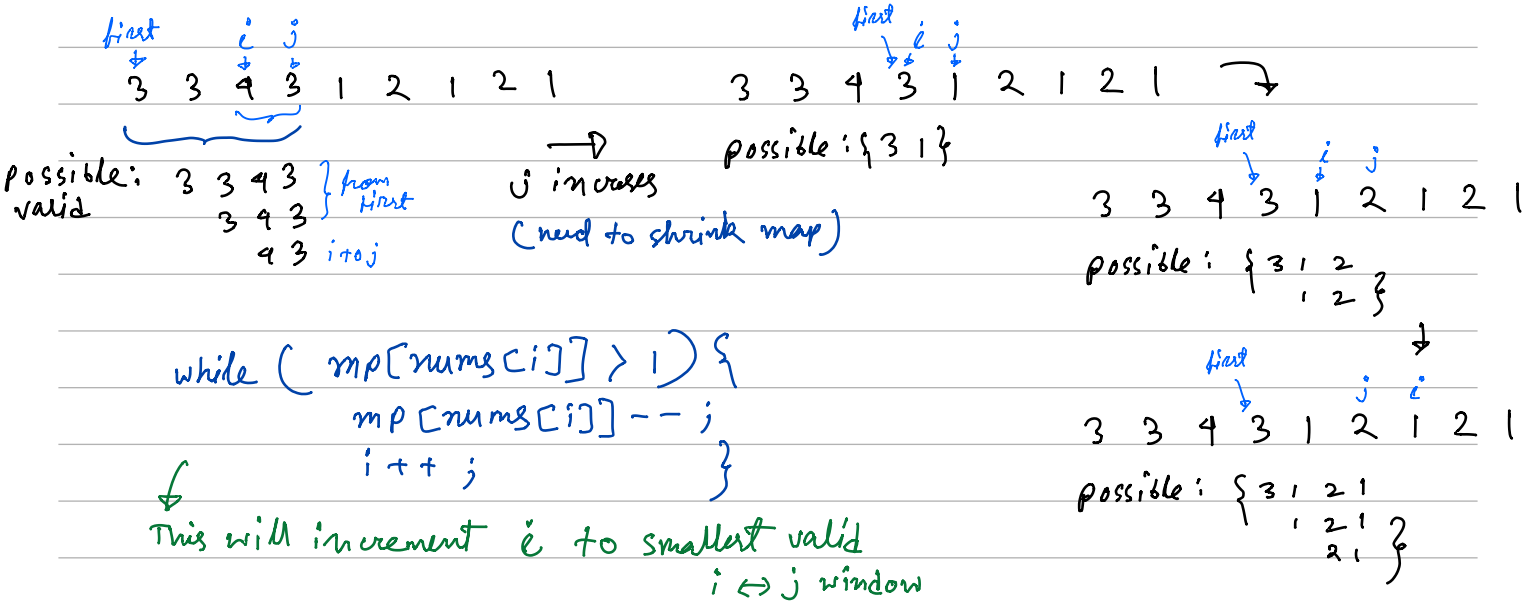
| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| | 1 | 2 | 1 | 2 | 3 | 4 |
| | i | i | i | i | i | i |
| Table: | 1:1 | 1:1<br>2:1 | 1:2 1<br>2:1 | 1:1<br>2:2 1 | ~~1:1~~<br>2:1<br>3:1 | ~~2:1~~<br>3:1<br>4:1 |
| | | i at 0 | i at 1 | i at 2 | i at 3 | i at 4 |
| ans: | | +1+(0) | +1+(1) | 1+(2) | +1+(0) | +1+(0) |
| += 1+(first i - i) | | {1 2} | {2 1} {1 2 1} | {1 2} {1 2 1 2} {2 1 2} | {2 3} | {3 4} |

accounting for

ans = 8 ✓

... 3 4 3 1 2 1 2 1 → smallest window of 2 distinct elements

whole window of those 2 distinct elements.

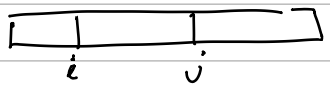



```
while ( mp[nums[i]] > 1 ) {
    mp[nums[i]] -- ;
    i++ ;
}
```

This will increment i to smallest valid i ↔ j window

```
while (j < n){
    mp[nums[j]] ++ j
    while (mp.Size() > K)
        mp[ nums[i] ] -- i  if 0 then erase
        i++ j
        first_i = i,
    while ( mp[nums[i]] > 1 )
        mp[nums[i]] -- j
        i++ j
    if (mp.size() == K) ans += 1 + (i - firsti)
    j++ j
}
```

## Longest Contiguous Subarray with abs(max - min) <= limit

approach 


multiset < pair < int, int > > → storing { nums[j], j }

while ( setMax->first - setMin->first > limit )
    st.erase ( { nums[i], i } ) ; i++
    // update setMax , setMin .

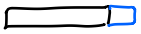
## Find Subarray with Bitwise And closest to k.

[ a b c d .. ]  find min | k - AND(subarray) |

x1 x2 x3 x4 x5

AND (x1 x2 x3 x4) ≤ AND (x1 x2 x3 x4 x5) (Always)

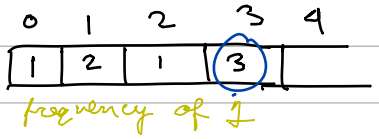
so,

if (K < AND)  j++ (expand) 

elseif (K > AND)  i++ (shrink) 

Now updating the AND without reiterating the array

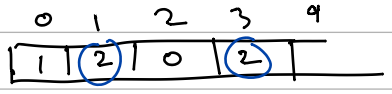
1 1 0 0  a1
1 0 1 0  a2
1 0 1 1  a3
1 0 0 0




3 1

remove a1 ,

1 0 1 0  a2
1 0 1 1  a3
2 0 1 0




3 1

len(subarr) == freq of 1
means No 0 at 1st and 3rd bit

# Count Substring that Satisfy K-Constraint - II

1 0 1 1 0 0 1 0 1 1 0 0 0 1

in the substring either count[0] <= K or count[1] <= K } then valid.

also the query count # valid substring from [low, high].

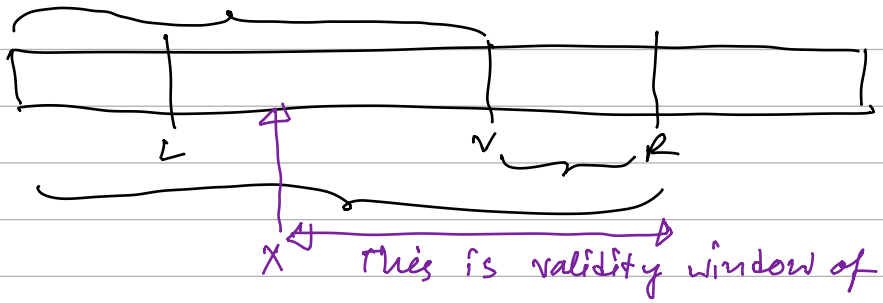
idea:


maximum valid window from low towards high.

len = validRightmost - low + 1

result = len x (len + 1) / 2

result += remaining.




This is validity window of R  L to R is not a valid window.

PrefixSumTill[R] = 0 to X + L1 x (L1 + 1)/2

PrefixSumTill[V] = 0 to L + L x (L+1)/2

---

V to R