Algorithms

17. Sliding Window

Handwritten by pankaj kumar

Sliding Window

Date

Page No. 245.

Date
Page No. 246.

① Introduction :-

Flow :- ① Origin of sliding window
② Brute
③ ~~Identify~~
④ Types of sliding window

problem :- arr = 2 3 5 2 9 7 1

k = 3 → size of subarray (continuous part of array)

2 3 5 2 9 7 1

2 3 5 → 10 (51)
3 5 2 → 10 (sum)
5 2 9 → 53
2 9 7 → 54
9 7 1 → 55

sum

O/P :- find max^m sum

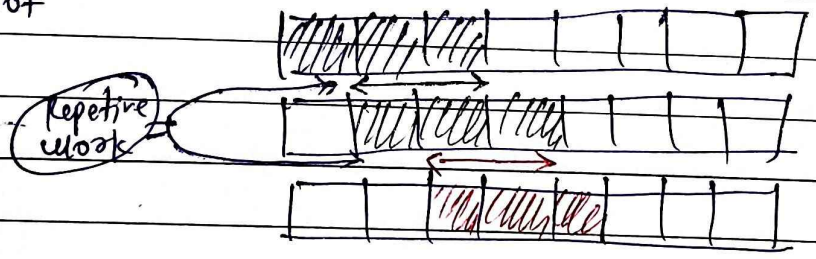
- **Brute force**

using loop

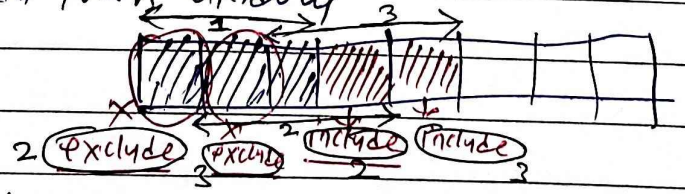
```
for (i = 0; i < n-k; i++) {
    for (j = i; j < i+k; j++) {
        sum += arr[j];
    }
}
```

- **Optimization**

- For optimization we have to check there is repetative work or not



- So to optimize this we can add next element and subtract first element from window



- This it's known as sliding window bc we are sliding window next one by one.

* Identify :-

1. we have given array / string
2. subarray / substring
3. largest / maxm / minm
4. k = window size

(if this is given then we can think about sliding window)

Note:- also there can be a situation where we have not given window size. and we have to find that window size

* Types of sliding window

fixed (easy)

- we have given fixed size of window and we slide one by one window and remove previous when window size increase.

variable (medium)

- we have not given window size. we have to find window size.
- it can be of type of find largest or, smallest window question
- and we have given a condition

variable size sliding window

ex:- arr: 3 2 4 5 1 1 1 1 1 3 3

Q. find largest subarray which sum is equal to 5.

[3 2] [4] [5] [1 1 1] [1 1] [3] 3

o/p:- 5, {1, 1, 1, 1, 1}

(2) Sliding window problem :-

Problems

(fixed)

1. max/min sub array of size k
2. 1st -ve in every window size of k
3. count occurrence of anagram
4. max of all subarray of size k
5. max of min for every window size

(variable)

1. largest/smallest subarray with sum k
2. largest sub-string with k distinct character
3. length of largest substring with no repeating characters
4. pick toy
5. minimum window substring.

Date
Page No. 248.

(3) maxm sum sub-array of size k:-

- we have given an array and a window size 'k', we have to find sub-array of size 'k' which sum of element is maxm.

ex:- arr[]: [2 | 5 | 1 | 8 | 2 | 9 | 1], k = 3

8, 14, 11, 19, 12

O/P :- 19

(*) Brute Force :-

generate all subarrays of size k, compute their sum and finally return the maxm of all sums. TC: O(n*k)

(*) Using sliding window :-

```
int maxSum (int arr[], int n, int k)
    if(n<k) return -1;
    int ans=0;
    for(int i=0; i<k; i++)
        ans += arr[i];
    int curr_sum = ans;
    for(int i=k; i<n; i++){
        curr_sum += arr[i]-arr[i-k];
        ans = max(ans, curr_sum);
    }
    return ans;
}
// TC: O(n)   SC: O(1)
```

(compute sum of first window of size k)

(compute sum of remaining windows by removing first element of previous window and adding last element of current window)

(4) First -ve integer in every window of size k:-

arr[]: 12 -1 -7 8 -15 30 16 28

k: 3

O/P:- [-1, -1, -7, -15, -15, 0]

Approach:- [12 | ## | -7 | 8 | -15 | 30 | 16 | 28]

queue / list :- [## | -7 | -15]

(Point first element)

[-1, -1, -7 .....

- traverse the array, when -ve number will be approach then put it into list, and for every window the first element will be first -ve integer of list in array
- when the -ve element from sliding window will be deleted then delete it also from list
- for a window if the size of list will be zero i.e list will be empty then for that window o/p will be zero.

Code :-

```
vector<int> FirstNegative(vector<int> A, int n){
    queue<int> q;
    vector<int> ans;
    int i=0, j=0;
    while(j<N){
        if(A[j]<0) q.push(A[j]);
        if(j-i+1<k) j++;
        else if(j-i+1==k){
            if(q.size()!=0){
                if(A[i]==q.front()){
                    ans.push_back(q.front());
                    q.pop();
                }else{
                    ans.push_back(q.front());
                }
            }else{ans.push_back(0);}
            i++; j++;
        }
    return ans;
}
```

// TC: O(n)
// SC: O(n)

Date
Page No. 250.

(5) Count Occurrences of Anagrams :-

- Given a word 'pat' and a text 'txt'. Return the count of the occurrences of anagrams of the word in the text.
- 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.

ex:-

txt = forxxorfxdofor
pat = for (anagrams = for, fro, ofr, orf, rof, ror)
O/P: 3 { For, orf and ofr appears in the txt, hence answer is 3 }

*Brute Force Approach :- traverse from start of the string considering substrings of length equal to the length of the given word and then check if this substring has all the characters of words. TC: O(n1*n2)

*Sliding window Approach :-

Method 1 :- (Youtube Tech dose)

```
int FindAnagrams (string p, string t) {
    int t_len = t.length();
    int P_len = P.length();
    if (t_len < P_len) return 0;
    vector<int> Freq_p (26,0);
    vector<int> window (26,0);
    int count=0;
    for (int i=0; i<P_len; i++) {          // first window
        Freq_P[p[i]-'a']++;
        window[t[i]-'a']++;
    }
    if (Freq_p == window)
        count++;
```

```
For (int i=p_len; i<t_len; i++){
    window[t[i-p_len]-'a']--;
    window[t[i]-'a']++;
    if (freq_p == window) count++;
}
return count;
}
```

// T.C: O(n*26) = O(n), n=t.len
// S.C: O(26) = O(1)

(*) Method 2: (Youtube: Aditya Verma):-

```
int findAnagram(string pat, string txt){
    map<char, int> mp;
    for(int i=0; i<pat.length(); i++) mp[pat[i]]++;
    int k = pat.length(); //size of window
    int st=0, end=0; //st=start window, end=end window
    int count=mp.size(); int ans=0;
    while(end<txt.length()){
        if(mp.find(txt[end])!=mp.end()){
            mp[txt[end]]--;
            if(mp[txt[end]]==0) count--;
        }
        if(end-st+1<k) end++;
        else if(end-st+1==k){ //window hitting case
            if(count==0) ans+=1;
            if(mp.find(txt[st])!=mp.end()){
                mp[txt[st]]++;
                if(mp[txt[st]]==1) count++;
            }
            st++; end++;
        }
    }
    return ans;
}
```

T.C: O(n)
S.C: O(1) · (O(26))

(6) Maximum of all subarrays of size k / (sliding window maximum)

ex:- arr[] = {1, 2, 3, 2, 4, 5, 2, 3, 6}, k = 3

o/p = [3, 3, 4, 5, 5, 5, 6]

(*) Naive Approach :- by using nested loop.

- First loop (external loop) will be run from starting index (0) to (n-k)
- and second (nested loop) will find the maxm (element) in index 'i' to index 'i+k'.

TC: O(n*k) / SC: O(1)

(*) Using AVL Tree (self balancing BST) :-

(1) create a self-balancing BST (AVL tree) to store and find maxm element.
(2) Tranverse over array. insert element in AVL tree
(3) if the loop counter is greater than or equal to k then delete (i-k)th element from the BST.
(4) print the maxm element of the BST.

TC:- O(N*logk) ; Insertion, deletion and search takes logk time in AVL trees. so overall time complexity is O(N*logk)

SC:- O(k)

(*) Sliding window + monotonic deque :-

- A question arises here is, "do we need to keep all the window element in deque" ?
  - an important observation is for two element arr[left] & arr[right], where left < right, arr[left] leaves the window earlier as we slide. if arr[right] > arr[left], then there is no point to keep arr[left] in deque since arr[right] is allways gonna be larger during the time arr[left] is in the window.
- monotonic deque :- deque with property that element from head to tail are in decreasing order.

and we can achieve this property by modifying push operation • Before we push an element into the deque, we first pop everything smaller than it out of the deque.

Steps to solve :-

(1) create a deque to store k elements. Run a loop and insert the first 'k' element in the deque. Before inserting check if the element at the end of queue is smaller than the current element, if it is then remove the element from end until all element left in deque are greater than current.

(2) Now, run a loop from k to end of array.

(3) Print front of deque (maxm of window).

(4) remove front element of deque if they are going out from current window.

(5) insert next element as inserted in step (1).

Code :-

```
vector<int> maxSlidingWindow(vector<int>& arr, int k){
    deque<int> dq;
    vector<int> ans;
    for(int i=0; i<arr.size(); i++){
        if(!dq.empty() && dq.front()==i-k)
            dq.pop_front();
        while(!dq.empty() && arr[dq.back()]<arr[i])
            dq.pop_back();
        dq.push_back(i);
        if(i>=k-1) ans.push_back(arr[dq.front()]);
    }
    return ans;
}
```

TC: O(n) SC: O(k)

Ex:- arr[] = {1, 2, 3, 1, 4}

(1) dq = 1

(2) arr[] = {1, 2, 3, 1, 4} dq = 2

(3) arr[] = {1, 2, 3, 1, 4} dq = 3 O/P: 3

(4) arr[] = {1, 2, 3, 1, 4} O/P: 3, 3 dq = 3, 1

(5) arr[] = {1, 2, 3, 1, 4} dq = 1, 4 O/P: 3, 3, 4

(7) Largest subarray of sum k (variable size sliding window)

I/P :- arr[] = {10, 5, 2, 7, 1, 9}, k = 15.

O/P :- 4, (The largest subarray is {5, 2, 7, 1})

(*) Naive Approach :- consider the sum of all the sub-arrays and return the length of the longest sub-array having the sum 'k'. TC :- O(n²).

```
int maxlen=0;
for (i=0 to n){
    int sum=0;
    for (j=i to n){
        sum += arr[j];
        if (sum==k){
            maxlen = max(maxlen, j-i+1);
        }
    }
}
return maxlen;
```

(*) Using Hash map :-

(i) initialize sum=0 and maxlen=0

(ii) create a hash table having (sum, index) tuples.

(iii) for i=0 to n-1 perform the following steps:-

→ Accumulate arr[i] to sum.

→ if sum==k, update maxlen = i+1.

→ Check whether sum is present in the hash table or not. if not present, then add it to the hash table as (sum, i)

→ check if (sum-k) is present in the hash table or not. if present, then obtain index of (sum-k) from the hash table as index. Now check if [maxlen < (i-index)], then update [maxlen = (i-index)]

(iv) return maxlen.

TC : O(N)

SC : O(N)

Date
Page No. 255

```
int lenOfLongSubarr(int arr[], int n, int k){
    unordered_map<int,int> um;
    int sum=0, maxLen=0;
    for(int i=0; i<n; i++){
        sum += arr[i];
        if(sum==k) maxlen = i+1;
        if(um.find(sum)==um.end()) um[sum]=i;
        if(um.find(sum-k)!=um.end()){
            if(maxlen<(i-um[sum-k]))
                maxlen = i-um[sum-k];
        }
    }
    return maxlen;
}
```

this step is basically we have to find longest subarrays

(*) Using variable size sliding windows :-

Note :- this won't work for negative numbers.

Approach :- initialize i, j, and sum = 0. ① if the sum is less than k add arr[j] to sum, just increment j, ② if sum is equal to k compute the maxlen by comparing with (j-i) if ③ the sum is greater than k subtract the ith element (arr[i]) & i++, while the sum is less than k.

```
int lenOfLongSubarr(int arr[], int n, int k){
    int i=0, j=0, sum=0, maxlen = INTMIN;
    while(j<N){
        sum += arr[j];
        if(sum<k){
            j++;
        }
        else if(sum==k){
            maxlen = max(maxlen, j-i+1);
            j++;
        }
        else if(sum>k){
            while(sum>k){
                sum -= arr[i];
                i++;
            }
            if(sum==k) max(maxlen, j-i+1);
            j++;
        }
    }
    return maxlen;
}
```

// TC : O(N)
// SC : O(1)

Date
Page No. 256.

atmost

(8) Longest substring with ^k unique characters.

(i) I/P :- str = "aaabbcc" , k = 1
O/P :- 2
Explanation :- max substring with 'k' unique character are {"aa", "bb", "cc"}

(ii) I/P :- str = "aaabbcc" , k = 2
O/P :- 4
Explanation :- {"aaabb", or, "abbcc"}

(*) Brute Force (Method 1) :-

(1) generate all the substring $O(n^2)$, and check on each for k unique character $O(n)$. so - total TC : $O(n^3)$.

(2) we can further optimize it by creating a hash table and while generating the substring, check the no. of unique character using that hash table TC : $O(n^2)$.

(*) Using Sliding window + hashing (method 2) :-

Algorithm Steps :-

(i) Create a hash table to store a mapping b/w characters and their frequency.

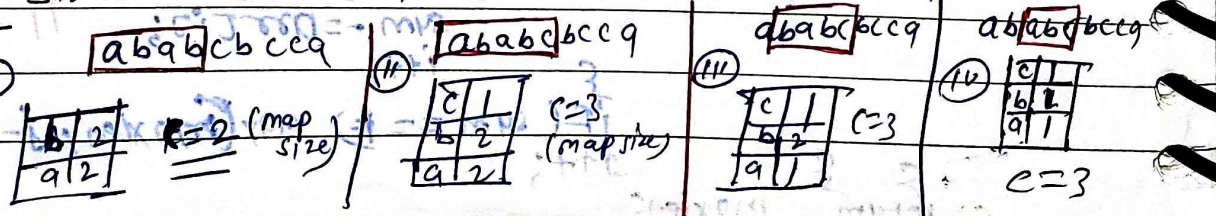
(ii) Insert new character from the string until we have 'k' distinct characters in the window. and expand the window from the right by adding new element till size is less than or equal to 'k'.
unique characters

(iii) if at any time no. of unique character in window reaches more than 'k', shrink the window from left side and reduce frequency of the character moving out until we have only k unique characters.

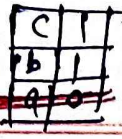
(iv) whenever the character frequency in hash table reaches 0, we will remove the character.

(v) After every expansion and shrinking, check if the current window has the largest size we have seen and if so, remember it.

ex :- str = ababbccca, k = 2



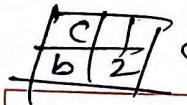
a b a b c b c c q



remove

(vi) now again expand from right until unique = 2,

a b a b c b c c q



Date

Page No. 257.

Code :-

```
int kDistinct (string str, int k) {
    unordered_map<char, int> table;
    int start=0, end=0;
    int longest=0;
    for (end=0; end< str.size(); end++) {
        table[str[end]]++;
        while (table.size()>k) {
            table[str[start++]]--;
            if (table[str[start]]==0)
                table.erase(str[start]);
            start++;
        }
        longest=max(longest, end-start+1);
    }
    return longest;
}
```

TC :- O(n)

SC :- O(k)

(9) Longest substring with without Repeating character.

(1) Input: S = "abcabcbb"
O/P : 3
Explanation: "abc" = without repeating character / longest string.

(2) I/P: S = "bbbbbb"
O/P: 1

* Method 1: (O(n³)) :- we can consider all substrings one by one & check for each substring whether contains all unique character or not. Substring generating: O(n²) and for checking unique characters: O(n)

- we can optimize this by checking unique character when generating the substring. we can use hash-table to check.

* Method 2: O(n) :- using Sliding window without hashing :-

- expand the window in right and store the frequency of character in Hash table, when hash table frequency will be greater than '1' then shrink the window from left until frequency of that character become '1'
- take (check) length of window at every expansion & shrink.

Date

Page No. 258.

Code :-

```
int lengthOfLongestSubstring (string s){
    vector<int> chars(128);
    int left=0, right=0;
    int res=0;
    while (right<s.length()){
        chars[s[right]]++;
        while (chars[s[right]]>1){
            chars[s[left]]--;
            left++;
        }
        res = max(res, right-left+1);
        right++;
    }
    return res;
}
```

Note :- We can optimize more by storing last index of a character instead of increasing Frequency. So that we don't need to shrink from left one by one we can direct goes to that location from where no duplicate will be encounter.

```
int lengthOfLongestSubstring (string s){
    vector<int> lastIdx(128,-1);
    int left=0, right=0, res=0;
    while (right<s.length()){
        // left = max(left, lastIdx[s[right]]+1);   // update left window
        res = max(res, right-left+1);
        // lastIdx[s[right]] = right;               // store index of character
        right++;
    }
    return res;
}
```

(10.) Minimum window Substring

Given two strings 's' and 't' of lengths 'm' and 'n' respectively, return the minimum window substring of 's' such that every character in 't' (including duplicates) is included in the window substring.

Note:- there can be extra character in window which is not in 't'.

ex:① I/P: s="ADOBECODEBANC", t="ABC"
O/P: "BANC" (the min window substring include 'A','B' & 'C' from t)

ex:② I/P: s="a", t="a", O/P: "a"

ex:③ I/P: s="a", t="aa", O/P: ""

code:- sliding window + hash table

```
string minWindow (string s, string t){
    unordered_map<char, int> m;        (or, alternate: vector<int> m(128,0);)
    for (auto c:t) m[c]++; //count of char in t
    int start=0, end=0, counter=t.size(), minStart=0, minLen=INT_MAX;
    int size=s.size();
    while (end<size){                  (most important line for shrinking the window in while loop)
        if (m[s[end]]>0) counter--;     (if char in s exist in t, decrease counter)
        m[s[end]]--; end++;             (if char not in t, m[s[end]] will be negative.)
        while (counter==0){ //(when we find a valid window move start to find smaller window)
            if(end-start<minLen){
                minStart=start;
                minLen=end-start;
            }
            m[s[start]]++;              (since we are removing/shrinking the window so increase the value which was decrease before)
            if (m[s[start]]>0) counter++;   (when value will be >0 means that character was in 't' so increase counter)
            start++;
        }                               (The counter will be 0 until all character will be removed from window which may not in 't'.)
    }
    return minLen==INT_MAX? "":s.substr(minStart, minLen);
}
```

TC: O(n) | SC: O(n)