SLIDING WINDOW

Converting a nested for loop solution into a single loop to reduce the time complexity

Notes completely based on

Aditya Verma - Sliding Window Playlist

Accepted solutions : GitHub - Ankit Kumar

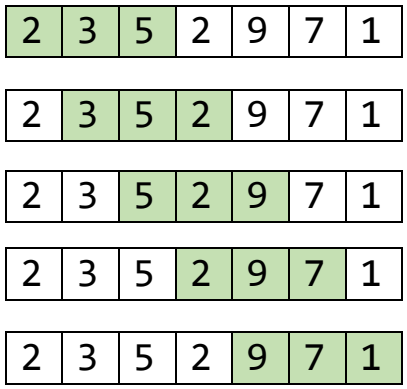
THERE IS NO SUCH THING AS "SLIDING WINDOW" IT IS JUST A WAY TO USE TWO POINTERS EFFICIENTLY AND SMARTLY

Flow:

1. Origin of Sliding Window
2. Brute force
3. Identification
4. Types of Sliding Window

Input:

```
Input: array[] = {2, 3, 5, 2, 9, 7, 1}
        window size (k) = 3
```



Naïve approach:

Time complexity – $O(n^2)$

```
for(int i=0; i<n-k; i++)
{
    for(int j=i; j<k; j++)
    {
        // some code and calculations
    }
}
```

Sliding Window Approach:

Time complexity – O(n)

```
int begin=0; // Denotes where the window begins
int end = 0; // Denotes where the window ends
while(end != array.size())
{
   while(end < k)
   {
     // Increasing end pointer until window size is hit.
        end++;
   }
   else
   {
     // Reached the window size
     // Our calculations
        begin++;
     // Move the window further
   }
}
```

Identification:

- We will be given an array or string.
- Asked about subarray, subsequence or substring
- A window size, or a condition for deciding the window size will be given.

Types of Sliding Windows:

1. Fixed size sliding window : Window size will be given in the question.
2. Variable size sliding window : Conditions will be given and we have to calculate the size of the window, satisfying the given condition(s).

Problems:

1. Fixed size sliding window :
    a. Max/min sub array of size k
    b. First negative in every window of size k
    c. Count occurrence of anagrams
    d. Max of all subarrays of size k
    e. Max of minimums for every window of size k

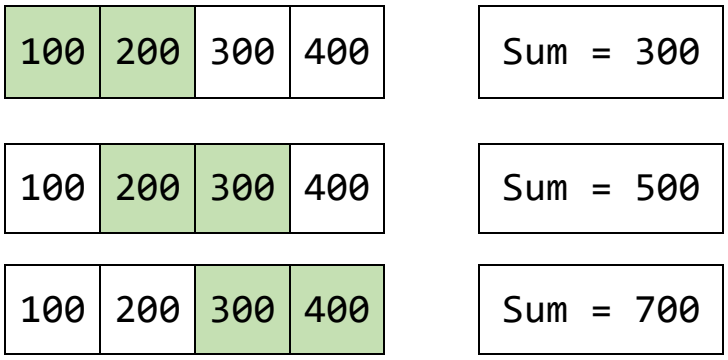
2. Variable size sliding window :
    a. Largest/Smallest subarray of sum k
    b. Largest substring with k distinct characters
    c. Length of largest substring with no repeating characters
    d. Pick Toy
    e. Minimum window substring

Q. – Easy Q. – Medium Q. – Hard

Q : [GFG - Max Sum Subarray of size K]

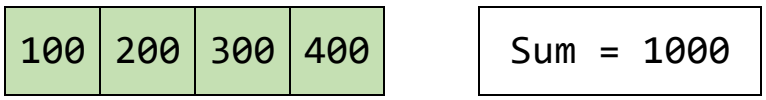
Given an array of integers array of size **N** and a number **K**. Return the maximum sum of a subarray of size K.

```
Input : N = 4, K = 2
        array = [100, 200, 300, 400]
```




```
Output : max(300, 500, 700) = 700
```

```
Input : N = 4, K = 4
        array = [100, 200, 300, 400]
```




```
Output : max(1000) = 1000
```

Solution:

Brute force:

```
int maxx = 0;
for(int i = 0; i<n-k; i++)
{
     int sum = 0;
     for(int j=i; j<k; j++)
          sum += array[j];
     maxx = max(maxx, sum);
}
```

Sliding Window:

```
int MaxSum(int k, vector<int> &nums , int N){
    int sum = 0;
    for(int i=0; i<k; i++)
         sum += nums[i];
    int maxx = sum;
    for(int i=k; i<nums.size(); i++)
    {
        sum += nums[i]-nums[i-k];
        maxx = max(maxx, sum);
    }
    return maxx;
}
```

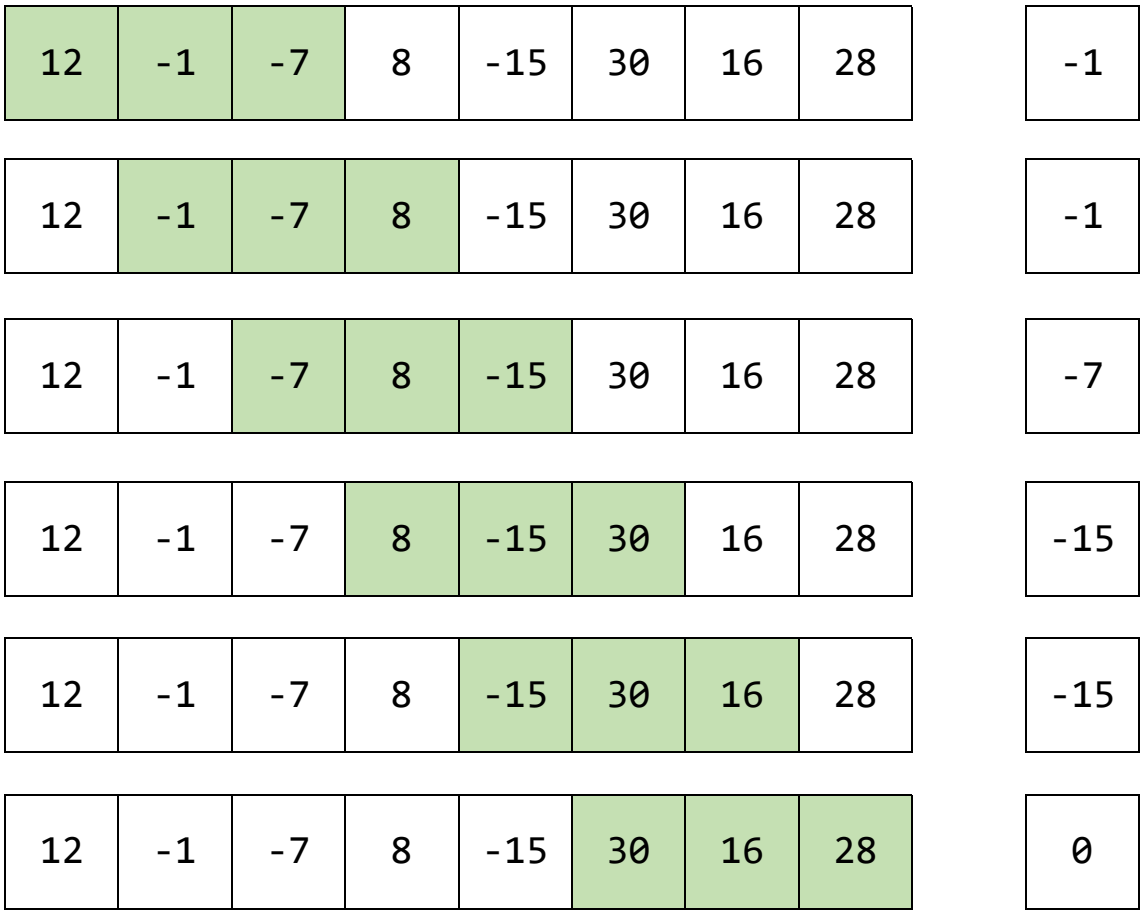
Q. [GFG - First negative integer in every window of size k]

Given an array **array[]** of size **N** and a positive integer **K**, find the first negative integer for each and every window(contiguous subarray) of size **K**.

```
Input : N = 8, K = 3
        array[] = [12, -1, -7, 8, -15, 30, 16, 28]
for no negative characters in the window return 0.

Output : [-1, -1, -7, 8, -15, 30, 16, 28]
```

In this question we will store the indices of the negative numbers that our current window has in a separate data structure, and when we will move our window one step forward, we will check if the element we just moved out from the left side is same as the first element from our negative array, if they are the same then we will have to pop that element out of our negative array, as its no more a part of the window, and push the first element from our negative array into the answer vector. If at any point we found an empty negative array, we will push zero for that window knowing that subarray doesn't have any negative integers. The size of our answer array will be N-K+1.



Solution:

```
vector<int> FirstNegative( int nums[], int N, int k)
{
    deque<long long> neg;
    vector<long long> ans;
    int j = 0;
    for(int i=0; i<k; i++)
    {
        if(nums[i]<0)
            neg.push_back(i);
    }
    if (neg.empty()) ans.push_back(0);
    else ans.push_back(nums[neg.front()]);
    for(int i=k; i<N; i++)
    {
        if(nums[i]<0) neg.push_back(i);
        if(!neg.empty() && i-k == neg.front()) neg.pop_front();
        if (neg.empty()) ans.push_back(0);
        else ans.push_back(nums[neg.front()]);
    }
    return ans;
}
```

Q. GFG - 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.

Anagram is word or phrase formed by rearranging the letters of a different words.

```
Input: txt = forxxorfxdofr
       pat = for
Output: forxxorfxdofr, forxxorfxdofr, forxxorfxdofr
        count(for, orf, ofr) = 3
```

Solution:

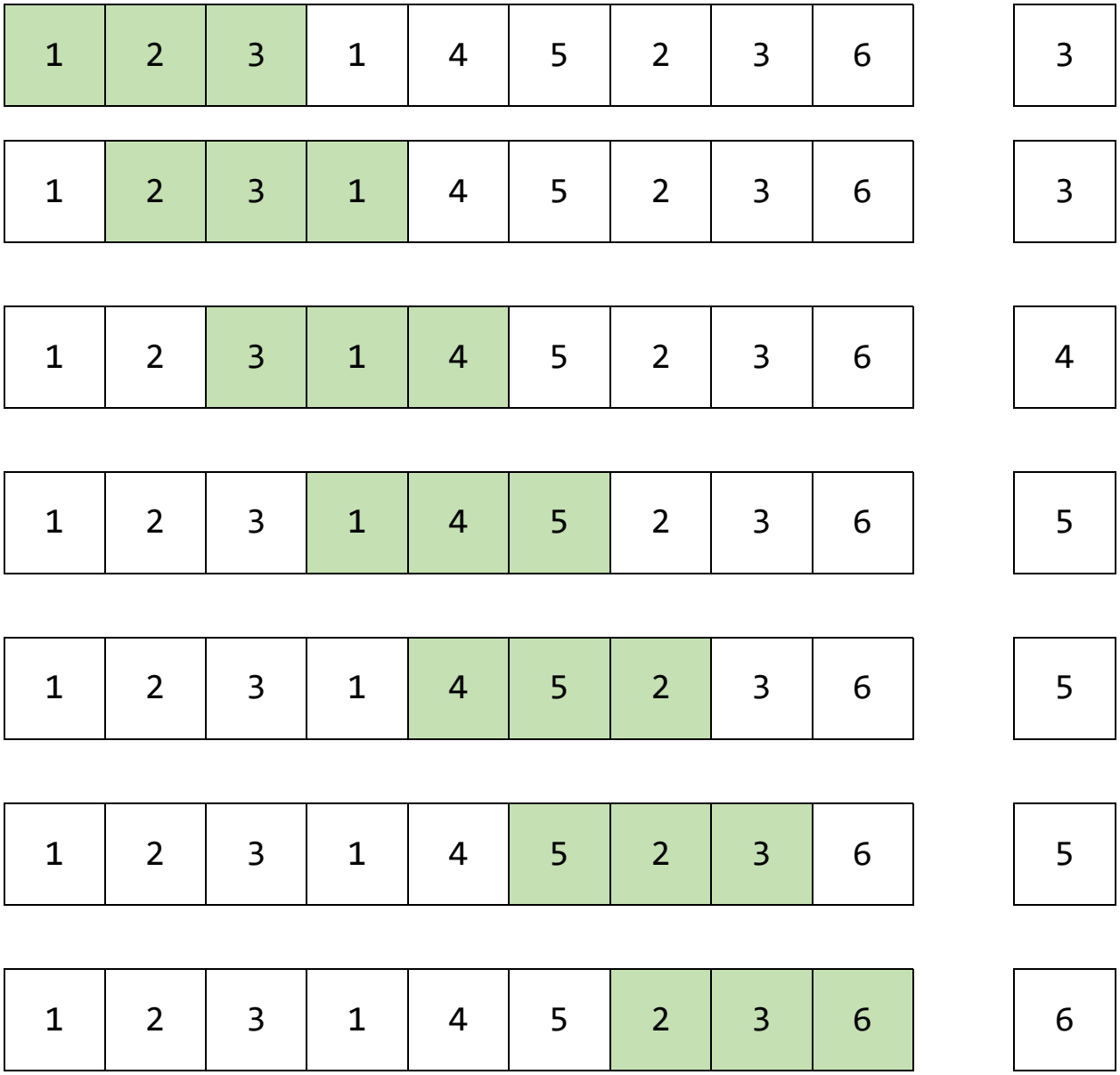
```
class Solution{
public:
    int search(string pat, string txt)
    {
        vector<int> cmp(26, 0), save(26, 0);
        for(int i=0; i<pat.size(); i++)
        {
            cmp[txt[i]-'a']++;
            save[pat[i]-'a']++;
        }
        int count = 0;
        if(save==cmp) count++;
        for(int i = pat.size(); i < txt.size(); i++)
        {
            cmp[txt[i]-'a']++;
            cmp[txt[i-pat.size()]-'a']--;
            if(save == cmp) count++;
        }
        return count;
    }
};
```

Q. GFG - Maximum of all subarrays , Leetcode : Sliding Window Maximum

Given an array array[] of size N and an integer K. Find the maximum for each and every contiguous subarray of size K.

```
Input : N = 9, K = 3
        array[] = [1, 2, 3, 1, 4, 5, 2, 3, 6]
Output : [3, 3, 4, 5, 5, 5, 6]
```

Store only the elements that are greater than our queue's front, so that we will know for sure that the front of queue always has the maximum for the current sliding window.



Solution:

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

Q. Leetcode - Longest Substring Without Repeating Characters

Given a string s, find the length of the **longest substring** without repeating characters.

```
Input : string = "abcabcbb"
Output : "abc"
```

Use a map or a queue to know what size of current substring.

Solution:

Using queue

```
class Solution {
public:
    int lengthOfLongestSubstring(string s)
    {
        deque<char> Q;
        int ans = 0;
        for(int i=0; i<s.size(); i++)
        {
            while(count(Q.begin(), Q.end(), s[i])!=0)
                Q.pop_front();
            Q.push_back(s[i]);
            if(Q.size()>ans) ans = Q.size();
        }
        return ans;
    }
};
```

Without using a HashMap or Queue:

```
class Solution {
public:
    int lengthOfLongestSubstring(string s)
    {
        int i=0;
        int j=1, int res=0;
        if(s.length()==1) return 1;
        while(i>=0 && j<s.length() && i<=j)
       {
           if(s[i]==s[j])
           {
               i++;
               j++;
           }
            else
            {
                for(int temp=i; temp<j; temp++)
                    if(s[temp]==s[j]) i = temp+1;
               j++;
            }
            res=max(res, j-i);
        }
        return res;
    }
};
```

Q. GFG - Longest K unique characters substring

Given a string you need to print the size of the longest possible substring that has exactly K unique characters. If there is no possible substring then print -1.

```
Input: S = "aabacbebebe", K = 3
    aabacbebebe & aabacbebebe (both strings have 3 unique chars)
Output: size("cbebebe") = 7
```

Solution:

```
class Solution {
  public:
    int longestKSubstr(string s, int k) {
        map<char, int> M;
        int j = 0, i = 0, ans = -1;
        while(j<s.size())
        {
            M[s[j]]++;
            if(M.size()==k) ans = max(ans, j-i+1);
            else if(M.size()>k)
            {
                while(M.size()>k)
                {
                    M[s[i]]--;
                    if(M[s[i]]==0) M.erase(s[i]);
                    i++;
                }
            }
            j++;
        }
        return ans;
    }
};
```

Q. Leetcode - Fruit Into Baskets aka Pick Toy

John is asked to pick any number of toys from a given of toys, given that he has to follow these restrictions.

a. Can pick only two types of toys, with unique IDs represented as integers.
b. Has to pick toys as a continuous subset from the given toys.

```
Input: toys = [1,2,3,2,2]
```

[1 2 3 2 2] & [1 2 3 2 2] two choices with two unique toys. Same as the previous question with K unique characters.

```
Output: 4
```

Solution:

```
int totalFruit(vector<int>& fruits) {
     map<int, int> M;
     int ans = 0, j = 0;
     for(int i=0; i<fruits.size(); i++)
     {
         M[fruits[i]]++;
         if(M.size()>2)
         {
             while(M.size()>2)
             {
                 M[fruits[j]]--;
                 if(M[fruits[j]]==0) M.erase(fruits[j]);
                 j++;
             }
         }
         ans = max(ans, i-j+1);
     }
     return ans;
 }
```

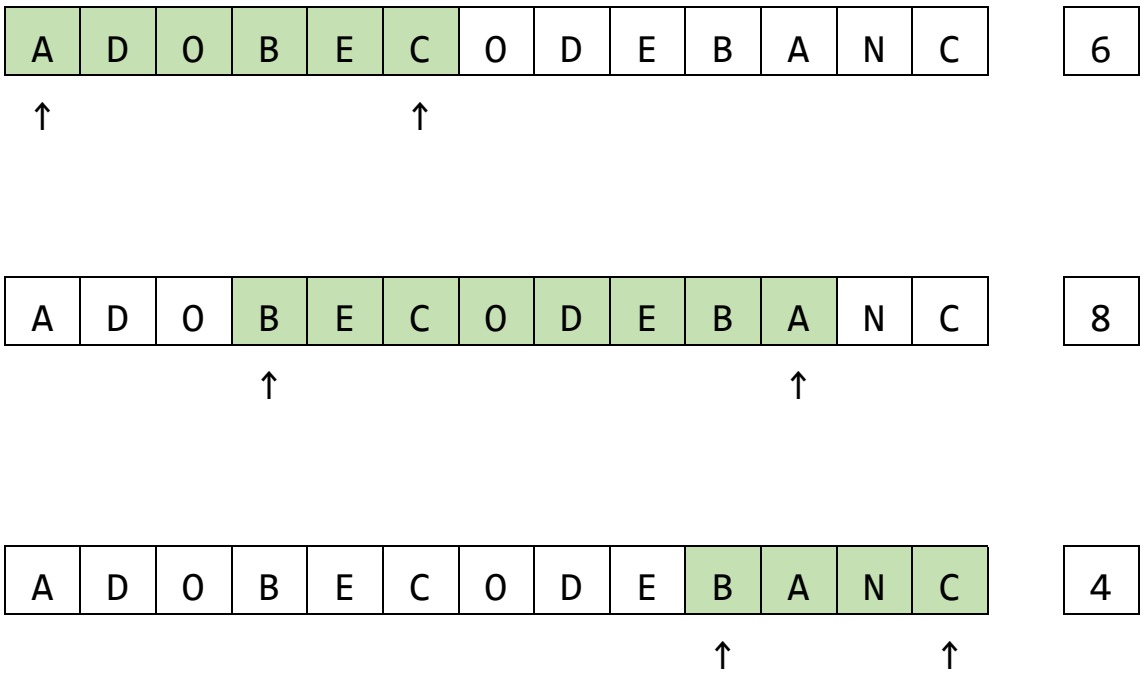
THE SLIDING WINDOW PROBLEM

Q. Leetcode - 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. If there is no such substring, return the empty string* "".

A **substring** is a contiguous sequence of characters within the string.

```
Input: s = "ADOBECODEBANC", t = "ABC"
```




```
Output : min (6, 8, 4) = "BANC"
```

Solution:

```
class Solution {
public:
    string minWindow(string s, string t)
    {
        vector<int>v(130,0),tm(130,0);
        for(auto &c:t) tm[c]++;
        int start = 0, length = INT_MAX, i = 0, j = 0, count = 0;
        while(j<s.size())
        {
            if(v[s[j]]<tm[s[j]])
                count++;
            v[s[j++]]++;
            while(count==t.size())
            {
                if(j-i<length)
                {
                    length=j-i;
                    start=i;
                }
                if(v[s[i]]==tm[s[i]])
                    count--;
                v[s[i++]]--;
            }
        }
        if(length==INT_MAX)
            return "";
        return s.substr(start, length);
    }
};
```