



csgator

Follow

Jan 9, 2018 · 6 min read

## Leetcode Pattern 2 | Sliding Windows for Strings

A fellow redditor from /r/cscareerquestions pointed me to this awesome thread on leetcode discuss which reveals the sliding window pattern for solving multiple string (substring) problems. Today we'll explore the intuition behind this powerful technique and apply it to some very famous string problems. Another fact I aim to prove is that some problems tagged as hard are really easy once we grasp the required pattern.

. . .

I strongly believe in learning by doing, so let's walk through a super simple example to understand how sliding windows work and then come back to leetcode problems. A sliding window approach generally helps us reduce the time complexity for brute force approaches.

Given an array of integers of size 'n'.

Our aim is to calculate the maximum sum possible for 'k' consecutive elements in the array.

Input : arr[] = {100, 200, 300, 400}  
k = 2

Output : 700

To solve it using brute force means to explore all possible cases. We really don't care about efficiency at this point but do want correct results. So we could run a nested loop exploring all windows of length k in the given array and pick the max sum.

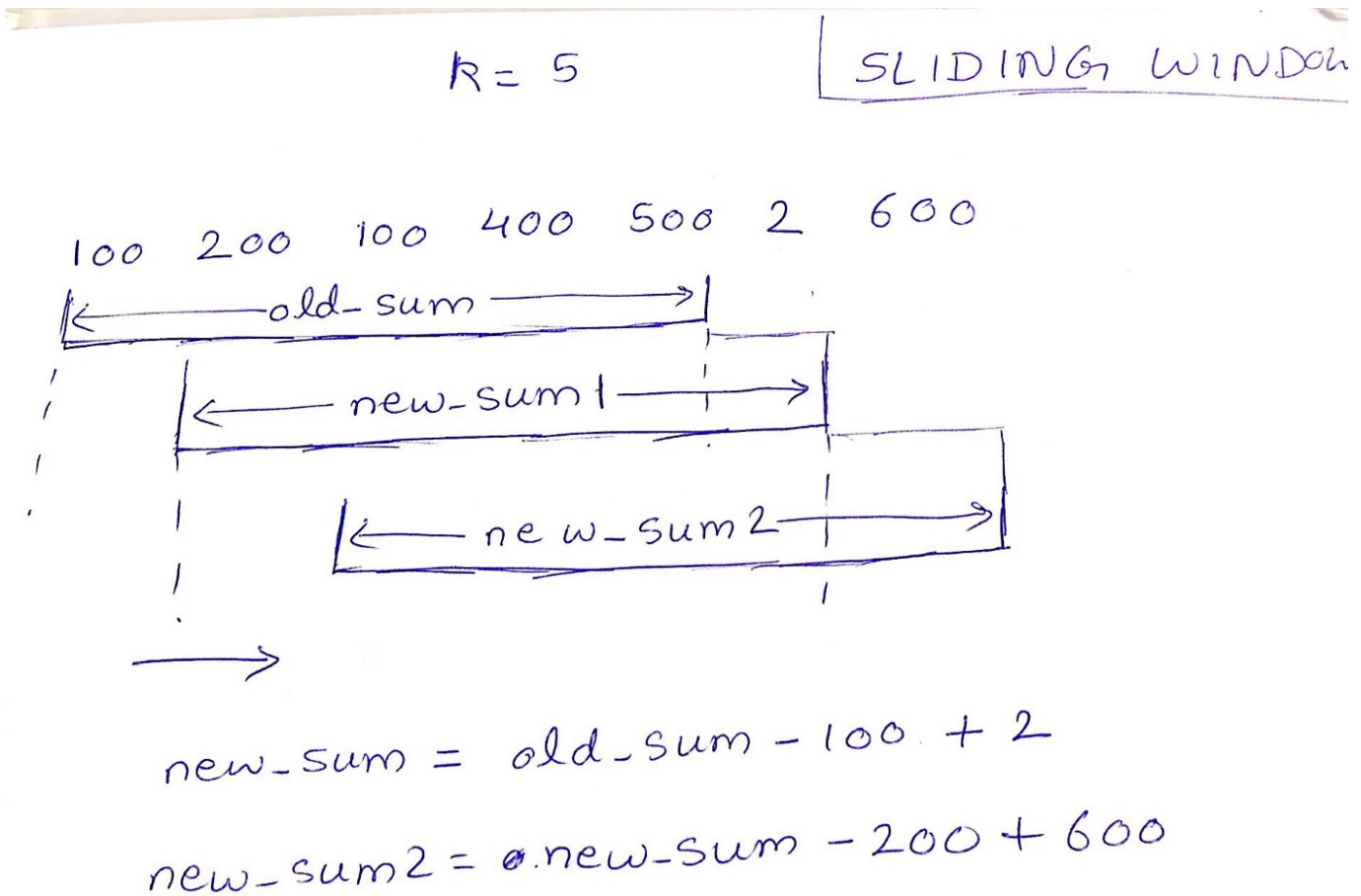
```
// BRUTE FORCE : iterate through all windows of size k

for(int i = 0; i < n-k+1; i++){
    int current_sum = 0;

    for(int j = 0; j < k; j++){
        current_sum = current_sum + arr[i+j];
    }

    max_sum = max(current_sum, max_sum);    // pick maximum sum
}
```

Now notice that having computed the sum of 1st window (size k), in order to get the sum of the next overlapping window we just need to leave out the leftmost item value and add the new (rightmost) item's value, so we are essentially saving the re-computation of the sum for the non-changing part of the window.



```

int max_sum = 0, window_sum = 0;

/* calculate sum of 1st window */
for (int i = 0; i < k; i++) window_sum += arr[i];

/* slide window from start to end in array. */

for (int i = k; i < n; i++){
    window_sum += arr[i] - arr[i-k];    // saving re-computation
    max_sum = max(max_sum, window_sum);
}

```

Voila ! we used memory to save time, a very classic trade-off in algorithms and sliding windows are all about this as we'll see. Time complexity reduced from  $O(n^2)$  to  $O(n)$ . Now let's get to actual leetcode problems.

. . .

## 76. Minimum Window Substring

*note : this first problem has been brutally analyzed as this is a tricky concept and forms the basis to solve the next 5. Master this one and try the rest on your own, that would be the best way to learn.*

In the previous example the window was of a fixed size, but here we use a window of variable size determined by *begin* and *end* markers. A brute force approach would be to iterate through all possible substrings and determine the minimum window which contains all characters of T. How do you see if a substring has all characters of T ? You could use the frequency table of T which stores character to number of occurrences as follows :

```

# initialize frequency table
for char c in T do
    table[c]++;
end

counter = table.size()           # unique chars in T

```

```

for char c in string B do
    if(char c in table){
        table[c]--;
        if(table[c] == 0) counter--;
    }
end

if(counter == 0){ # B has every character in T }

```

So we are basically assuring that every unique character in T exists in B as many times as it exists in T by maintaining a counter. It is fine if there are 4 'K's in T and B has 7 'K's , the table count for 'K' would simply go negative, but it goes to 0 at some point before that, proving string B has at least 4 'K's in it, satisfying the need with respect to "K", extending the logic to other chars in T if counter = 0, B has all chars in T.

Okay so coming to the sliding window part here, we keep sliding *end* to the right examining each new character and updating the count in table. As soon as we hit counter = 0, it means we have a valid answer so we try to trim it down removing the unessential characters from the start by sliding begin right. We constantly keep trying to validate / invalidate the string by manipulating counter and table counts.

Take a moment, understand this code. Walk through it on paper for this example : [ S : ADOBECODEBANC | T : "ABC" ]. Do not worry, the code is just heavily annotated, it is actually very concise.

```

1  class Solution {
2  public:
3      string minWindow(string s, string t) {
4          unordered_map<char, int> table;
5
6          // initialize frequency table for t
7          for(char c : t){
8              table[c]++;
9          }
10
11         // initialize sliding window
12         int begin = 0, end = 0;
13         int counter = table.size();
14         int len = INT_MAX;
15
16         string ans = "";

```

```
7
8 // start sliding window
9 while(end < s.length()){
0     char endchar = s[end];
1
2     // if current char found in table, decrement count
3     if(table.find(endchar) != table.end()){
4         table[endchar]--;
5         if(table[endchar] == 0) counter--;
6     }
7
8     end++;
9
0     // if counter == 0, means we found an answer, now try to trim that window by s
1     while(counter == 0){
2         // store new answer if smaller than previously best
3         if(end-begin < len){
4             len = end - begin;
5             ans = s.substr(begin, end - begin);
6         }
7
8         // begin char could be in table or not,
9         // if not then good for us, it was a wasteful char and we shortened the pr
0
1         // if found in table increment count in table, as we are leaving it out of
2         // so it would no longer be a part of the substring marked by begin-end wi
3         // table only has count of chars required to make the present substring a
4         // if the count goes above zero means that the current window is missing o
5         int startchar = s[begin];
6
7         if(table.count(startchar) == 1){
8             table[startchar]++;
9             if(table[startchar] > 0) counter++;
0         }
1
2         begin++;
3     }
4 }
5
6 return ans;
7 }
8 };
```

Intuition : the best substring for the answer would simply be a permutation of T if such a substring exists in S, but otherwise we could have wasteful characters sitting in between the essential characters that make the substring valid as an answer. Our attempt here is to remove such chars without losing the necessary ones. After trimming down as much as possible we resume with sliding *end* right and repeating the whole process.

S = ADOBECODEBANC  
T = ABC

frequency table for T →  $\left\{ \begin{array}{cc} A & 1 \\ B & 1 \\ C & 1 \end{array} \right\}$

sliding window:  $\text{end} = 0, \text{begin} = 0$   
 $\text{counter} = \text{table.size}() = 3$

1)  $\begin{array}{c} \text{begin} \rightarrow \\ \downarrow \\ \text{A} \text{ D O B E C O D E B A N C} \\ \uparrow \\ \text{end} \rightarrow \end{array}$   $\left\{ \begin{array}{cc} A & 0 \\ B & 1 \\ C & 1 \end{array} \right\}$  counter 2

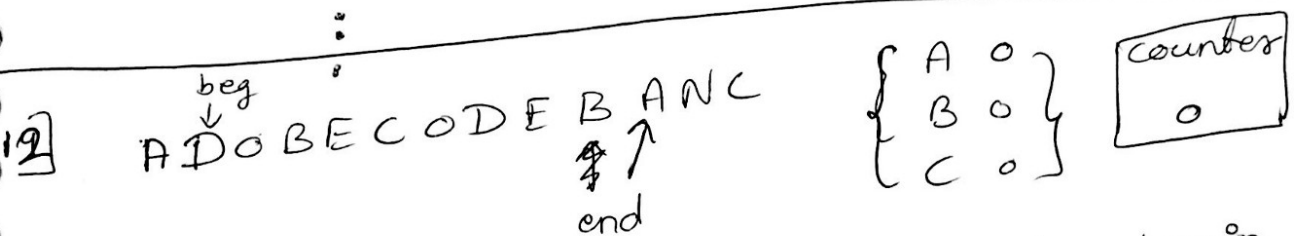
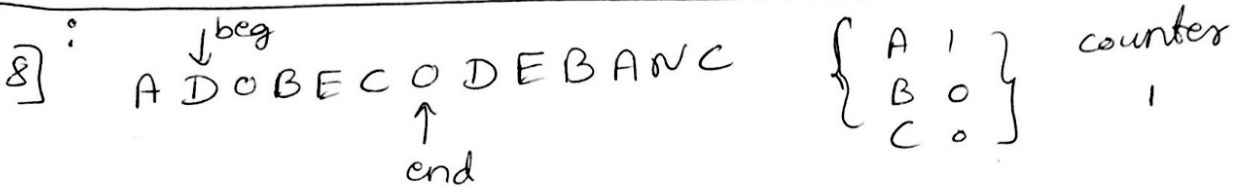
2)  $\begin{array}{c} \text{begin} \\ \downarrow \\ \text{A} \text{ D O B E C O D E B A N C} \\ \uparrow \\ \text{end} \end{array}$   $\left\{ \begin{array}{cc} A & 0 \\ B & 1 \\ C & 1 \end{array} \right\}$  counter 2

3)  $\begin{array}{c} \text{begin} \\ \downarrow \\ \text{A} \text{ D O B E C O D E B A N C} \\ \uparrow \\ \text{end} \end{array}$   $\left\{ \begin{array}{cc} A & 0 \\ B & 0 \\ C & 1 \end{array} \right\}$  counter 2

4)  $\begin{array}{c} \text{begin} \\ \downarrow \\ \text{A} \text{ D O B E C O D E B A N C} \\ \uparrow \\ \text{end} \end{array}$   $\left\{ \begin{array}{cc} A & 0 \\ B & 0 \\ C & 1 \end{array} \right\}$  counter 1



if counter == 0 | valid answer |  
try minimizing the answer  
slide begin and see if some non-essential  
characters can be dropped.



try minimizing answer by sliding begin.



counter = 0



counter = 0

*end*

Whenever counter = 0 we have a valid candidate for our ans, but we update ans only if it is shorter than previously recorded minimum length ans.

### 438. Find All Anagrams in a String

Exactly the same as above with the added condition that the substring should be of length equal to p and that we have to return indexes of all such occurrences.

```
1  class Solution {
2  public:
3      vector<int> findAnagrams(string s, string p) {
4          unordered_map<char, int> table;
5          vector<int> ans;
6
7          for(char c : p){
8              table[c]++;
9          }
10
11         if(s.length() < p.length() || s.length() == 0) return ans;
12
13         int begin = 0, end = 0, word_size = p.length();
14         int counter = table.size();
15
16         while(end < s.length()){
17             char endchar = s[end];
18
19             if(table.count(endchar) == 1){
20                 table[endchar]--;
21                 if(table[endchar] == 0) counter--;
22             }
23
24             end++;
25
26             while(counter == 0){
27                 if(end - begin == word_size) {
28                     ans.push_back(begin);
29                 }
```



```

30
31         char beginchar = s[begin];
32
33         if(table.count(beginchar) == 1){
34             table[beginchar]++;
35             if(table[beginchar] > 0) counter++;
36         }
37
38         begin++;
39     }
40 }
41
42 return ans;
43 }
44 };

```

AnagramsInString.cpp hosted with ❤ by GitHub

[view raw](#)

### 30. Substring with Concatenation of All Words

Here comes my proof that not all Leetcode “hard” are exactly hard, this one is simply a slight modification of the above problem which is tagged “easy”. Instead of chars in above question now we have words so it got a bit messier.

```

1  class Solution {
2  public:
3      vector<int> findSubstring(string s, vector<string>& words) {
4          unordered_map<string, int> table;
5          vector<int> ans;
6
7          if(words.size() == 0) return ans;
8
9          int window_size = 0;
10         int word_size = words[0].length();
11
12         // building frequency table
13         for(string word : words){
14             window_size += word.length();
15             table[word]++;
16         }
17
18         unordered_map<string, int> reference(table);
19
20         int begin = 0, end = 0, counter = table.size();

```

```
20 int begin = 0, end = 0, counter = table.size(),
21 vector<string> tokens;
22
23 if(s.length() < window_size) return ans;
24
25 // we increment begin and end only in word_size
26 // there are only word_size possible start points for our window.
27 // end is actually the start of the last word in window or put in other words
28 // the real end of window is actually at end + word_size
29 for(int i = 0; i < word_size; i++){
30     begin = i; end = i;
31     table = reference; // reset to original frequency table after every iteration
32     counter = table.size();
33
34     while(end + word_size - 1 < s.length()){
35         string lastword = s.substr(end, word_size);
36
37         if(table.count(lastword) == 1){
38             table[lastword]--;
39             if(table[lastword] == 0) counter--;
40         }
41
42         if(end + word_size - begin == window_size){
43             // counter == 0, valid answer !
44             if(counter == 0){
45                 ans.push_back(begin);
46             }
47
48             string firstword = s.substr(begin, word_size);
49
50             if(table.count(firstword) == 1){
51                 table[firstword]++;
52                 if(table[firstword] > 0) counter++;
53             }
54
55             begin += word_size;
56         }
57
58         end += word_size;
59     }
60 }
61
62 return ans;
63 }
64 };
```

30-Substring\_Concatenation\_of\_All\_Words.cpp hosted with ❤ by GitHub

[view raw](#)

### 3. Longest Substring Without Repeating Characters

Here we build the frequency table out of the substring we explored so far as the aim is to find longest substring without any repetitions. So similar to above case we slide *end* and keep building the table and as soon as we hit a repetition, meaning the *end char* is already in table, we start sliding *begin*. We record the max before sliding *begin* for the ans.

```
1  class Solution {
2  public:
3      int lengthOfLongestSubstring(string s) {
4          unordered_map<char, int> seen;
5
6          int begin = 0, end = 0;
7          int len = 0;
8
9          string ans = "";
10
11         while(end < s.length()){
12             char current = s[end];
13
14             if(seen.count(current) == 1 && seen[current] >= begin){
15                 begin = seen[current] + 1;
16             }
17             else{
18                 seen[current] = end;
19                 end++;
20             }
21
22             if(end - begin > len){
23                 len = end - begin;
24                 ans = s.substr(begin, end - begin);
25             }
26         }
27
28         return len;
29     }
30 };
```

LongestSubstringWithoutRepeatingChars.cpp hosted with ❤ by GitHub

[view raw](#)

## 159. Longest Substring with At Most Two Distinct Characters

```
1  class Solution {
2  public:
3      int lengthOfLongestSubstringTwoDistinct(string s) {
4          if(s.length() == 0) return 0;
5
6          unordered_map<char, int> table;
7          int begin = 0, end = 0, len = 0, counter = 0;
8
9          while(end < s.length()){
10             char current = s[end];
11
12             table[current]++;
13             if(table[current] == 1) counter++;
14
15             end++;
16
17             while(counter > 2){
18                 char startchar = s[begin];
19
20                 if(table.count(startchar) == 1){
21                     table[startchar]--;
22                     if(table[startchar] == 0) counter--;
23                 }
24
25                 begin++;
26             }
27
28             len = max(len, end - begin);
29         }
30
31         return len;
32     }
33 };
```

159-Longest\_Substring\_with\_At\_Most\_Two\_Distinct\_Characters.cpp hosted with ❤ by GitHub

[view raw](#)

## 567. Permutation in String

```
1  class Solution {
2  public:
3      bool checkInclusion(string s1, string s2) {
```

```
4      unordered_map<char, int> table;
5
6      for(char c : s1){
7          table[c]++;
8      }
9
10     int begin = 0, end = 0, counter = table.size();
11
12     while(end < s2.length()){
13         char endchar = s2[end];
14
15         if(table.count(endchar) == 1){
16             table[endchar]--;
17             if(table[endchar] == 0) counter--;
18         }
19
20         end++;
21
22         while(counter == 0){
23             if(end - begin == s1.length()) return true;
24
25             char startchar = s2[begin];
26
27             if(table.count(startchar) == 1){
28                 table[startchar]++;
29                 if(table[startchar] > 0) counter++;
30             }
31
32             begin++;
33         }
34     }
35
36     return false;
37 }
38 };
```

567-Permutation\_in\_String.cpp hosted with ❤ by GitHub

[view raw](#)

Other problems that can be solved similarly:

**340. Longest Substring with At Most K Distinct Characters** which is literally the same as the one with  $K = 2$ .

## 424. Longest Repeating Character Replacement

A good exercise at this point would be to think why the sliding window approach actually works, draw out all the cases possible for the minimum window substring problem and you'll have a deeper intuition into this technique.

Some interesting facts :

1. It took me less time to actually solve these problems than what it took me to figure out how to explain it in writing.
2. I solved a problem tagged hard first and the corresponding easy problem later as I use an extension to hide the difficulty levels on leetcode lol.
3. I strongly believe understanding how a programmer wrote code is more important than the code itself and so I am working on codeback , a code playback tool allowing users to stop, edit and play the code line by line as well as execute the code to see output.


Stay tuned and feel free to connect with me on LI :


<https://www.linkedin.com/in/sourabh-reddy>

[Programming](#) [Leetcode](#) [Algorithms](#) [Tech](#) [Interview](#)

[About](#) [Help](#) [Legal](#)

Get the Medium app

 A button that says 'Download on the App Store', and if clicked it will lead you to the iOS App store

 A button that says 'Get it on, Google Play', and if clicked it will lead you to the Google Play store