76. Minimum Window Substring

<https://leetcode.com/problems/minimum-window-substring/>

1. **Listen**

**Problem Statement:**

Given two strings s and t of lengths m and n respectively, return the***minimum window substring***ofs such that every character int(***including duplicates***) is included in the window. If there is no such substring, return the empty string"".

The testcases will be generated such that the answer is **unique**.

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

**Input:**

1. string s with length m
2. string t with length n

**Goal:**

find the minimum substring in s that contains all letters (including duplicates) from t

**Return:**

return the minimum window substring of s such that every character int(**including duplicates**) is included in the window.

return an empty string “” if there is no such substring.

1. **Examples**

**Example 1:**

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

**Output:** "BANC"

**Explanation:** The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.

**Example 2:**

**Input:** s = "a", t = "a"

**Output:** "a"

**Explanation:** The entire string s is the minimum window.

**Example 3:**

**Input:** s = "a", t = "aa"

**Output:** ""

**Explanation:** Both 'a's from t must be included in the window.

Since the largest window of s only has one 'a', return empty string.

**Constraints:**

* m == s.length
* n == t.length
* 1 <= m, n <= 105
* s and t consist of uppercase and lowercase English letters
* The answer is **unique**.

**Test Cases:**

* s and t are the same string
* t is a single letter
* t does not appear anywhere in s (return empty string)
* varying substrings t that appear in s

1. **Brute Force / Optimization / Walkthrough**

**Solution 1: Time = O(T + S), Space = O(T)**

**Determining the Constraint**

Essentially, our loop constraint needs to be whether all of the characters of the substring t are present in our current window (of string s).

**Global and Local Tracker Variables**

This means we need at least four variables

1. Two global pointers: start and end to keep track of minimum substring window.
2. Two local pointers: start and end to keep track of current substring window

// s = "ADOBECODEBANC", t = "ABC"

public String minWindow(String s, String t)

{

// global window substring pointers

int **subStart** = 0, **subEnd** = 0;

// local window substring pointers

int **start** = 0, **end** = 0;

}

**Character Frequency**

* To check if a window is valid, we use an integer array to store letter frequency.
* Since we have TWO substrings and are checking if one string is a substring of another, we will need to count the frequency of each character in the shorter substring (usually, the shorter string is the t instead of the s.
* Basically, we are loading up the character frequency of the t string, so we can find the first window in s that contains all the letters in t.

// s = "ADOBECODEBANC", t = "ABC"

public String minWindow(String s, String t)

{

int[] **letterFrequency** = new int[128];

for(char c : t.toCharArray()) **letterFrequency**[c]++;

// global window substring pointers

int **subStart** = 0, **subEnd** = 0;

// local window substring pointers

int **start** = 0, **end** = 0;

}

* Here, as you can see, we have a frequency array of all the characters in the t substring.
* The characters from substring **t** in the **letterFrequency** array can either have
  + positive values (when initializing)
  + 0 value (when subtracting)
* If there are letters in string **s** that are not in substring **t**, then their indexes in **letterFrequency** can only have
  + negative values
  + 0 value
* The main intention of the frequency array is to
  + **ignore duplicate letters**

and

* + keep track of character frequency
* We will be iterating over the main string, s, and subtracting from the character frequency array **letterFrequency** each time we encounter a letter in s that is in **letterFrequency**.
* Therefore, we another global variable:

1. A character frequency array to detect when all the characters from t are present in s.

**Growing the Window**

* We can conveniently use a for loop to automatically increment the end pointer over the collection because we only need to iterate by 1 index for the outer loop.

// s = "ADOBECODEBANC", t = "ABC"

public String minWindow(String s, String t)

{

int[] **letterFrequency** = new int[128];

for(char c : t.toCharArray()) **letterFrequency**[c]++;

// global substring pointers

int **subStart** = 0, **subEnd** = 0;

// local window substring pointers

int **start** = 0;

for(int **end** = 0; **end** < s.length(); **end**++)

// do stuff

}

* Currently, the only values in **letterFrequency** with values > 0 are the indexes with letters from **t**.
* While growing the window, we decrement the frequency of each character we see in **s** from the **letterFrequency** array.
* For example, if **letterFrequency[‘A’]** = 2, that means the letter ‘A’ appears twice in t.
* Each time we run into the letter ‘A’ in s, we substract from letter frequency.
* Therefore, when **letterFrequency[‘A’]** reaches 0, that means two ‘A’s will be present in the current window.

**Satisfying the Constraint(s)**

* Once all the frequency of letters from **letterFrequency** reaches 0, this means that all the letters from **t** are present in the current window in **s** (meaning we have found a valid substring!).
* In order to make keeping track of the necessary letter frequencies easier (to know when all **letterFrequency** cells reach 0), we keep a tracker variable, **count**, that determines if we meet the problem condition (that all letters in t are currently in the window).
* **Count** represents the number of characters that we still need to find from **t** in our window.
  + When building letterFrequency, we iterate this counter up.
  + When looping through the s string, we iterate this counter down each time we run into a character that is present in letterFrequency.
* When the **count** variable reaches 0, that means all the characters from t are currently present in the substring window of s.
* Now, there may be shorter windows farther down string s, but we have found a current valid window that satisfies our condition.
* We now add a sixth global variable, count:

1. A counter to keep track of when all characters from t are present in s (just makes it easier to know when all cells in letterFrequency = 0).

// s = "ADOBECODEBANC", t = "ABC"

public String minWindow(String s, String t)

{

int **count** = t.length();

int[] **letterFrequency** = new int[128];

for(char c : t.toCharArray()) **letterFrequency**[c]++;

// global substring pointers

int **subStart** = 0, **subEnd** = 0;

// local window substring pointers

int **start** = 0;

for(int **end** = 0; **end** < s.length(); **end**++)

{

char **endChar** = s.charAt(**end**);

if(**letterFrequency**[**endChar**] > 0) **count**--;

**letterFrequency**[**endChar**]--;

}

// condition to update global trackers and shrink window

while(**count** == 0)

}

**Duplicates**

* Notice how this method specially handles duplicate characters.
* Every time we encounter a letter in the s substring window that is present in t, we decrement its value from **letterFrequency**.
* After filling out the characters counts in the **letterFrequency** array (at the very beginning of the function), the only indices that should have **positive character counts** are the characters present in the **t** string.
* Only when their counts are **less than or equal to zero (letterFrequency**[char] **<= 0)**, does that mean we have accounted for EVERY letter from the t string in the current window (includes duplicates).
* Notice how this aligns with the following statement:

**letterFrequency[endChar] > 0**.

* This ensures that ONLY the characters that are present in t get accounted for when updating the **count** variable.
* That way, when we enter the inner while loop when counter == 0, we can shrink the window while ignoring duplicates.
* If there are more of a certain character in the current window compared to the **t** string, then their **letterFrequency** value will continue to go negative, but will not affect the **count** variable, because we only update it when the frequency count of a character is > 0.
* For example: string s = "AAABECODEBANC" and t = "AABC"
  + After counting character frequencies for **letterFrequency**, **letterFrequency[‘A’]** will have a value of 2.
  + Every **if(letterFrequency[endChar]-- > 0) counter--;** statement we encounter decrements a character’s count.
  + Only the characters present in the **t** string that show up in the window will update the **count** variable, because the only characters in letterFrequency that can have positive values are the ones from **t**.
  + Therefore, when letterFrequency[‘A’] 0, that means **the same number of ‘A’s are present in the t string and the current window.**
  + When letterFrequency[‘A’] is less than 0, then that means there are more than necessary ‘A’ in the current window, and therefore does not affect the **count**.
  + Here is the original:

s = [AAABECODEBANC], t = AABC

**letterFrequency[‘A’]** = 2

**count** = 4

* + After updating the window past the first A:

s = A[AABECODEBANC], t = AABC

**letterFrequency[‘A’]** = 1

**count** = 3

* + Updating the window again:

s = AA[ABECODEBANC], t = AABC

**letterFrequency[‘A’]** is now 0

**count** = 2

* + Updating the window again:

s = AA[ABECODEBANC], t = AABC

**letterFrequency[‘A’]** = -1

**count** **is not affected**.

**Updating Global Variables**

* Now that we have satisfied our problem constraint, we must update our global variable trackers.
* We have decided on **two global variable trackers**
  + A pointer to the **front** of valid substring
  + A pointer to the **end** of the valid substring
* If our current VALID window is smaller than our previous VALID window, we update the pointer and length tracker variables.
* We check
  + the current minimum substring length (subEnd - subStart)

against

* + the current window length (end – start).
* We then try to shrink the window.

// s = "ADOBECODEBANC", t = "ABC"

public String minWindow(String s, String t)

{

int **count** = t.length();

int[] **letterFrequency** = new int[128];

for(char c : t.toCharArray()) **letterFrequency**[c]++;

// global substring pointers

int **subStart** = 0, **subEnd** = 0;

// local window substring pointers

int **start** = 0;

for(int **end** = 0; **end** < s.length(); **end**++)

{

char **endChar** = s.charAt(**end**);

if(**letterFrequency**[**endChar**] > 0) **count**--;

**letterFrequency**[**endChar**]--;

}

// condition to update global trackers and shrink window

while(**count** == 0)

{

if((**end** – **start**) < (**subEnd** – **subStart**)

{

**subStart** = **start**;

**subEnd** = **end**;

}

}

}

**Shrinking the Window**

* When we find our first valid window
  + **The characters in the t string will have a value of 0 in the letterFrequency array**.
  + **Every other character that is not present in the t string will have a negative value.**
* We have also updated out global variables to match the current valid window.
* Now, we need to tryand shrink the window.
  + After shrinking the window, the window may or may not still be valid (the problem constraint may or may not still be satisfied).
  + This means we would need to update our global variables again, and continue to shrink the array until our window is no longer valid (the problem constraint is no longer satisfied)
* We increment the character count for the letter at the start of the window.
* This negates the time we decremented its value outside the loop.
* Then, we check

if(++**letterFrequency**[**startChar**] > 0) count++;

* The characters from substring **t** are the only indexes can **ever have a positive value** in the **letterFrequency** array.
* Because in the outer loop, when we decremented each character count from the letterFrequency array using **if(letterFrequency[endChar]-- > 0) counter--;**, all characters that are not present in t will have negative values.
* While we are adding the values back to **letterFrequency**, their character count values will never go above 0, because we are just cancelling out their decremented values by incrementing the same amount.
* Therefore, **count** is only incremented when we encounter a character from **t**.
* That way, when we enter the inner while loop when counter == 0, we can shrink the window while ignoring duplicates, and exit the loop at the correct time.
* For example: string s = "AAADOBECODEBANC" and t = "ABC"
  + After building the **letterFrequency** array, **letterFrequency**[‘A’] will equal 1.
  + Once we iterate over the “AAA” in s, the value of **letterFrequency**[‘A’] will be -2.
  + After entering the while loop and updating the global variables, we increment the value of **letterFrequency**[‘A’] to -1.
  + We then re-enter the while loop, update global values, and update **letterFrequency**[‘A’] to -1.
  + Then again to 0.
  + Then again to 1.
  + Our current window will then look like this: "AAA[DOBECODEBANC]".
  + Notice, that since we no longer have all the characters from **t** inside our window, we no longer satisfy the loop condition, and thus are done shrinking the window, and go back to expanding it, looking for another A.
* We continue to shrink the window by bringing the start of the window forward until we reach a point that a **character from substring t is no longer present in the window** (which in this case, will be A).
* In order to actually shrink the window, each time we see a letter, we update the letter frequency array, and update the **start** pointer.
* We will reenter the loop if all letters from t are still present in the window.
* Therefore, when we are shrinking the window, it is possible that after shrinking, our new, smaller window is still valid, and we can update the global variables to reflect that.
* We continue to loop until we no longer satisfy the loop condition, and thus are done shrinking the window, and go back to expanding it.

**Visual Example**

s = “ADOBECODEBANC”, t = “ABC”

[ | A D O B E C O D E B A N C ] subStart = 0, subMax = 0

[ | A | D O B E C O D E B A N C ] subStart = 0, subMax = 0

[ | A D | O B E C O D E B A N C ] subStart = 0, subMax = 0

[ | A D O | B E C O D E B A N C ] subStart = 0, subMax = 0

[ | A D O B | E C O D E B A N C ] subStart = 0, subMax = 0

[ | A D O B E | C O D E B A N C ] subStart = 0, subMax = 0

[ | A D O B E C | O D E B A N C ] subStart = 0, subMax = 0

Enter inner loop

[ | A D O B E C | O D E B A N C ] subStart = 0, subMax = 5

[ A | D O B E C | O D E B A N C ] subStart = 0, subMax = 5

Exit inner loop

[ A | D O B E C O | D E B A N C ] subStart = 0, subMax = 5

[ A | D O B E C O D | E B A N C ] subStart = 0, subMax = 5

[ A | D O B E C O D E | B A N C ] subStart = 0, subMax = 5

[ A | D O B E C O D E B | A N C ] subStart = 0, subMax = 5

[ A | D O B E C O D E B A | N C ] subStart = 0, subMax = 5

Enter inner loop

[ A | D O B E C O D E B A | N C ] subStart = 0, subMax = 5

[ A D | O B E C O D E B A | N C ] subStart = 0, subMax = 5

[ A D O | B E C O D E B A | N C ] subStart = 0, subMax = 5

[ A D O B | E C O D E B A | N C ] subStart = 0, subMax = 5

[ A D O B E | C O D E B A | N C ] subStart = 0, subMax = 5

[ A D O B E C | O D E B A | N C ] subStart = 0, subMax = 5

Exit inner loop

[ A D O B E C | O D E B A | N C ] subStart = 0, subMax = 5

[ A D O B E C | O D E B A N | C ] subStart = 0, subMax = 5

[ A D O B E C | O D E B A N C | ] subStart = 0, subMax = 5

Enter inner loop

[ A D O B E C | O D E B A N C | ] subStart = 0, subMax = 5

[ A D O B E C O | D E B A N C | ] subStart = 0, subMax = 5

[ A D O B E C O D | E B A N C | ] subStart = 0, subLength = 5

[ A D O B E C O D E | B A N C | ] subStart = 9, subLength = 12

[ A D O B E C O D E B | A N C | ] subStart = 10, subLength = 12

return s.substring(9, 12 + 1)

This will return “BARC”

**Return**

Note that we must add + 1 when returning the substring.

return ((subEnd - subStart) == Integer.MAX\_VALUE ? "" : s.substring(subStart, subEnd+1));

The **s.substring(subStart, subEnd+1)** has the two global window variables in terms of indicies. In Java, the second argument of substring is exclusive, so we must add 1.

**Runtime**

We can fill the letterFrequency array in O(T) time, where T = t.length;

We will traverse the entire s string in O(S) time, where S = s.length;

In the worst case, we may end up visiting every element in string s twice (in case the subarray is at the very end of string s). This makes the loop run in O(2S) time.

Therefore, the total runtime is O(T + 2S) which equals O(T + S).

The space complexity is O(T) since we only keep a supplementary letterFrequency array of size t.length().

1. **Implement**

int findSubstring(string s){

// global tracker variables

int min/max;

// local trackers

int[] letterFrequency = new int[128];

for() { /\* initialize the hash map here if there is a second string \*/ }

int counter; // check whether the substring is valid

// pointers to start and end of window

int start = 0;

for(int end = 0; end < s.length(); end++)

{

// update local tracker

if(letterFrequency[s[end]] ?){ /\* modify counter here \*/ }

letterFrequency[s[end]]--;

// shrink left side of window until we no longer satisfy condition

while(/\* condition to shrink the window \*/){

/\* update global tracker if we satisfy condition\*/

/\* increase start to make condition invalid/valid again

letterFrequency[s[end]]++;

if(letterFrequency[s[end]] ?){ /\* modify counter here \*/

}

/\* update d here if finding maximum\*/

}

return d;

}

/\*

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 "".

The testcases will be generated such that the answer is unique.

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

\*/

import java.util.\*;

public class MinimumWindowSubstring {

public static void main(String[] args)

{

System.out.println(checkInclusion("ab", "eidboaoo"));

}

public static boolean checkInclusion(String s1, String s2) {

// keep count of number of chars from s2 in current window

int count = s1.length();

int[] frequency = new int[128];

for(char c : s1.toCharArray()) frequency[c]++;

// local trackers

int lStart = 0, lEnd = 0;

// traverse over s1

for(lEnd = 0; lEnd < s2.length(); lEnd++)

{

// grow window

char endChar = s2.charAt(lEnd);

if(frequency[endChar]-- > 0) count--;

// all characters from s1 are in s2

while(count == 0)

{

// if characters from s1 are a contiguous permutation in s2 return true

if((lEnd - lStart + 1) == s1.length()) return true;

// shrink window

char startChar = s2.charAt(lStart);

if(++frequency[startChar] > 0) count++;

lStart++;

}

}

return false;

}

public static String minWindow(String s, String t)

{

int [] map = new int[128];

for (char c : t.toCharArray())

{

map[c]++;

}

int start = 0, end = 0;

int minStart = 0, minLen = Integer.MAX\_VALUE;

int counter = t.length();

while (end < s.length())

{

/\*

Section 1

1. the map currently counts the number of all chars in substring t

2. we continue to loop until we find the valid subtring in s that has all of the chars in t

iterate over each char in string s. Subtract 1 from the map the chars we are currently looking at

(the substring chars will have values >= 0 in the map)

(chars that are in the string s and not in the substring t will have values < 0 in the map if we iterate over them before finding all chars in substring)

3. Add 1 to end to see the range in string s that all chars in string t are present

4. Once we find all the chars from t in s, the counter will reach 0, and we will enter the second loop

\*/

final char c1 = s.charAt(end);

if (map[c1] > 0) counter--;

map[c1]--;

end++;

/\*

Section 2 - We have found a valid window, so move start to find smaller window.

1. We save the window of indexes where we found the chars of t inside s

i.e. a first pass over s = "ADOBECODEBANC" t = "ABC" will yield start = 0 and end = 6

this means that the current minimum window that we can find the chars ABC inside s is between indexes 0 to 6

\*/

while (counter == 0) {

/\*

Only enter this condition if we find a smaller window that contains s in t

\*/

if (minLen > end - start)

{

minLen = end - start;

minStart = start;

}

/\*

Section 3

1. Continue to iterate over string s until you find the first occurence of some char from string t.

This is your new starting point to find another window that containts t in s.

2.

\*/

final char c2 = s.charAt(start);

map[c2]++;

if (map[c2] > 0) counter++;

start++;

}

}

return minLen == Integer.MAX\_VALUE ? "" : s.substring(minStart, minStart + minLen);

}

public String minWindow2s(String s, String t) {

// global target tracker

int minStart = 0, minLength = Integer.MAX\_VALUE;

// local tracker

int[] letterFrequency = new int[128];

for(char c : t.toCharArray()) letterFrequency[c]++;

int count = t.length();

// pointers to start and end of window

int start = 0;

for(int end = 0; end < s.length(); end++)

{

// update end of window

char endChar = s.charAt(end);

// the only values in letterFrequency with values > 0 are the indexes with letters from t (line 8)

// continue to iterate local tracker until we have found a window with all letters;

// once all the frequency of letters from t reach 0, it means we have found a valid substring

if(letterFrequency[endChar] > 0) count--;

// update letter frequency

letterFrequency[endChar]--;

// keep shrinking left side of window to see if we can satisfy the condition with a smaller window

while(count == 0)

{

// update global target tracker

if(minLength > end - start + 1)

{

minLength = end - start + 1;

minStart = start;

}

// shrink window size until we no longer satisfy the condition

char startChar = s.charAt(start);

letterFrequency[startChar]++;

// all letters from t should be have the value 0 in the letterFrequency map.

// this means that only letters from t can have positive values in letterFrquency.

// all other letters from string s should have negative values (other letters not in s are not incremented)

// the first time we find/pass a letter from t, we increment its frequency, meaning we have shrunk the window

// such that we no longer satisfy the condition of having all letters from t in s

if(letterFrequency[startChar] > 0) count++;

start++;

// the first time the incrementor start goes past a letter from t in s, we break from the loop because we no

// longer satisfy the condition. now we try to see if we can find a smaller substring satisfying the condition later

// in the string s.

// we will return to the loop once we satisfy the condition again, and then we will again continue to

// shrink the left side of the window until the condition is no longer satisfied

}

}

return (minLength == Integer.MAX\_VALUE ? "" : s.substring(minStart, minStart + minLength));

}

}

1. **Test**