LeetCode Training Day 22 String Coverage

Word coverage is a combination of two pointer sliding window and the hash table to count the characters in the substring. The problem normally will ask you what is the longest or shortest substring which can cover the specific number of characters.

The pattern is a slide window which is driven by a left pointer and right pointer, when right pointer moves, add character count, when left pointer move reduce character count.

## 3. Longest Substring Without Repeating Characters

Medium

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

**Example 1:**

**Input:** s = "abcabcbb"

**Output:** 3

**Explanation:** The answer is "abc", with the length of 3.

**Example 2:**

**Input:** s = "bbbbb"

**Output:** 1

**Explanation:** The answer is "b", with the length of 1.

**Example 3:**

**Input:** s = "pwwkew"

**Output:** 3

**Explanation:** The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

**Constraints:**

* 0 <= s.length <= 5 \* 104
* s consists of English letters, digits, symbols and spaces.

### Analysis:

When right pointer moves, we add character count, if none of the character count is more than 2, we keep track it as result.

Please note that the character set is not limited to lower case in English, so we cannot use vector to replace hash table.

/// <summary>

/// Leet code #3. Longest Substring Without Repeating Characters

/// Given a string, find the length of the longest substring without

/// repeating characters.

/// Examples:

/// Given "abcabcbb", the answer is "abc", which the length is 3.

/// Given "bbbbb", the answer is "b", with the length of 1.

/// Given "pwwkew", the answer is "wke", with the length of 3.

/// Note that the answer must be a substring, "pwke" is a subsequence

/// and not a substring.

/// </summary>

int LeetCodeString::lengthOfLongestSubstring(string s)

{

unordered\_map<char, int> char\_count;

int begin = -1;

int result = 0;

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

{

char\_count[s[end]]++;

while (char\_count[s[end]] > 1)

{

begin++;

char\_count[s[begin]]--;

}

result = max(result, end - begin);

}

return result;

}

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

Medium

Given a string s, return *the length of the longest substring that contains at most****two distinct characters***.

**Example 1:**

**Input:** s = "eceba"

**Output:** 3

**Explanation:** The substring is "ece" which its length is 3.

**Example 2:**

**Input:** s = "ccaabbb"

**Output:** 5

**Explanation:** The substring is "aabbb" which its length is 5.

**Constraints:**

* 1 <= s.length <= 105
* s consists of English letters.

### Analysis:

Use two pointers for sliding window to calculate characters, the size of hash table (number of keys) is the number of distinct characters. If we exceed more than 2 distinct characters, we move left point and reduce characters, when it becomes zero, we remove the key from hash table.

/// <summary>

/// Leet code #159. Longest Substring with At Most Two Distinct Characters

///

/// Given a string, find the length of the longest substring T that contains

/// at most 2 distinct characters.

/// For example, Given s = “eceba”,

/// T is "ece" which its length is 3.

/// </summary>

int LeetCodeString::lengthOfLongestSubstringTwoDistinct(string s)

{

unordered\_map<char, int> char\_count;

int begin = -1;

int result = 0;

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

{

char\_count[s[end]]++;

while (char\_count.size() > 2)

{

begin++;

char\_count[s[begin]]--;

if (char\_count[s[begin]] == 0)

{

char\_count.erase(s[begin]);

}

}

result = max(result, end - begin);

}

return result;

}

## 340. Longest Substring with At Most K Distinct Characters

Medium

Given a string s and an integer k, return *the length of the longest substring of* s *that contains at most* k ***distinct****characters*.

**Example 1:**

**Input:** s = "eceba", k = 2

**Output:** 3

**Explanation:** The substring is "ece" with length 3.

**Example 2:**

**Input:** s = "aa", k = 1

**Output:** 2

**Explanation:** The substring is "aa" with length 2.

**Constraints:**

* 1 <= s.length <= 5 \* 104
* 0 <= k <= 50

### Analysis:

Use two pointers for sliding window to calculate characters, the size of hash table (number of keys) is the number of distinct characters. If we exceed more than K distinct characters, we move left point and reduce characters, when it becomes zero, we remove the key from hash table.

/// <summary>

/// Leet code #340. Longest Substring with At Most K Distinct Characters

///

/// Given a string, find the length of the longest substring T that contains

/// at most k distinct characters.

/// For example, Given s = "eceba" and k = 2,

/// T is "ece" which its length is 3.

/// </summary>

int LeetCodeString::lengthOfLongestSubstringKDistinct(string s, int k)

{

unordered\_map<char, int> char\_count;

int begin = -1;

int result = 0;

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

{

char\_count[s[end]]++;

while (char\_count.size() > k)

{

begin++;

char\_count[s[begin]]--;

if (char\_count[s[begin]] == 0)

{

char\_count.erase(s[begin]);

}

}

result = max(result, end - begin);

}

return result;

}

## 424. Longest Repeating Character Replacement

Medium

You are given a string s and an integer k. You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most k times.

Return *the length of the longest substring containing the same letter you can get after performing the above operations*.

**Example 1:**

**Input:** s = "ABAB", k = 2

**Output:** 4

**Explanation:** Replace the two 'A's with two 'B's or vice versa.

**Example 2:**

**Input:** s = "AABABBA", k = 1

**Output:** 4

**Explanation:** Replace the one 'A' in the middle with 'B' and form "AABBBBA".

The substring "BBBB" has the longest repeating letters, which is 4.

**Constraints:**

* 1 <= s.length <= 105
* s consists of only uppercase English letters.
* 0 <= k <= s.length

### Analysis:

When we say replace characters to make a substring with only one distinct character, we are supposed to replace the minorities.

So we should only keep the slide window to track the characters most frequent, if the total of the minorities, which is the total length - most frequent character, exceeding k, we know the slide window is too long we have to shrink it by move the left point. When the substring within the window have the minority with less than or equal to k, we compare with result to get the maximum size.

/// <summary>

/// Leet code #424. Longest Repeating Character Replacement

///

/// Given a string that consists of only uppercase English letters,

/// you can replace any letter in the string with another letter at

/// most k times. Find the length of a longest substring containing

/// all repeating letters you can get after performing the above

/// operations.

/// Note:

/// Both the string's length and k will not exceed 10^4.

///

/// Example 1:

/// Input:

/// s = "ABAB", k = 2

/// Output:

/// 4

/// Explanation:

/// Replace the two 'A's with two 'B's or vice versa.

///

/// Example 2:

/// Input:

/// s = "AABABBA", k = 1

/// Output:

/// 4

/// Explanation:

/// Replace the one 'A' in the middle with 'B' and form "AABBBBA".

/// The substring "BBBB" has the longest repeating letters, which is 4.

/// </summary>

int LeetCodeString::characterReplacement(string s, int k)

{

vector<int> char\_count(26);

int result = 0;

int max\_count = 0;

int first = 0, last = 0;

while (last < (int)s.size())

{

char\_count[s[last]-'A']++;

// we only need to track the max count of characters

max\_count = max(max\_count, char\_count[s[last] - 'A']);

last++;

// if valid we track the length

if (max\_count + k >= last - first)

{

result = max(result, last - first);

}

else

{

// when invalid, we shrink window by one

char\_count[s[first] - 'A']--;

first++;

}

}

return result;

}

# Advanced Problems

## 395. Longest Substring with At Least K Repeating Characters

Medium

Given a string s and an integer k, return *the length of the longest substring of* s *such that the frequency of each character in this substring is greater than or equal to* k.

**Example 1:**

**Input:** s = "aaabb", k = 3

**Output:** 3

**Explanation:** The longest substring is "aaa", as 'a' is repeated 3 times.

**Example 2:**

**Input:** s = "ababbc", k = 2

**Output:** 5

**Explanation:** The longest substring is "ababb", as 'a' is repeated 2 times and 'b' is repeated 3 times.

**Constraints:**

* 1 <= s.length <= 104
* s consists of only lowercase English letters.
* 1 <= k <= 105

### Analysis:

This problem is hard level.

For this problem we have two methods to resolve, one is to divide and conquer, we count all characters. When we see a character with less than K repeating character, we split this string into left and right part, and do recursion.

But here we want to try another way, since we only have 26 characters, we can assume the result may have 1 distinct character, 2 distinct characters until 26. Then how about we keep slides window to maintain the distinct character from 1 to 26? Remember every time we see a character count from 0 to 1, we know there is a new distinct character, and we have 1 new character which may not meet the condition of at lease K count. But when we see this character count become K, we know we have 1 less character count not meet condition. All of this happened when we move right pointer and add character counts. On the other hand, when we move the left point and reduce the character count when it comes to k-1, we know we have 1 more character not meet condition, in this case no\_meet should add 1, but if the count reduced to zero and such character disappear, we should reduce no\_meet by 1.

/// <summary>

/// Leet code #395. Longest Substring with At Least K Repeating Characters

///

/// Find the length of the longest substring T of a given string

/// (consists of lowercase letters only) such that every character in T

/// appears no less than k times.

///

/// Example 1:

/// Input:

/// s = "aaabb", k = 3

/// Output:

/// 3

/// The longest substring is "aaa", as 'a' is repeated 3 times.

///

/// Example 2:

/// Input:

/// s = "ababbc", k = 2

/// Output:

/// 5

/// The longest substring is "ababb", as 'a' is repeated 2 times and 'b' is repeated 3 times.

/// </summary>

int LeetCodeString::longestSubstring(string s, int k)

{

int result = 0;

for (int i = 1; i <= 26; i++)

{

vector<int> char\_map(26);

int char\_count = 0;

int not\_meet = 0;

int begin = -1;

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

{

char\_map[s[end] - 'a']++;

if (char\_map[s[end] - 'a'] == 1)

{

char\_count++;

not\_meet++;

}

if (char\_map[s[end] - 'a'] == k)

{

not\_meet--;

}

while (char\_count > i)

{

begin++;

char\_map[s[begin] - 'a']--;

if (char\_map[s[begin] - 'a'] == k -1)

{

not\_meet++;

}

if (char\_map[s[begin]- 'a'] == 0)

{

char\_count--;

not\_meet--;

}

}

if (not\_meet == 0)

{

result = max(result, end - begin);

}

}

}

return result;

}

## 76. Minimum Window Substring

Hard

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.

**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.

**Follow up:** Could you find an algorithm that runs in O(m + n) time?

### Analysis:

This is my favorite problem, although it is hard. Same as before you need to keep a slide window to track the character count. First we need to count the original character for the word in a hash table, then we use this hash table to compare the substring in the slide window to the original word. When the character count from slide window reach to the same as target for each character, we know the character is covered. When all characters are cover we start to shrink the slide window, and deduct the character count when it is less than target count it become uncovered.

/// <summary>

/// Leet code #76. Minimum Window Substring

///

/// Given a string S and a string T, find the minimum window in S which

/// will contain all the characters in T in complexity O(n).

/// For example,

/// S = "ADOBECODEBANC"

/// T = "ABC"

/// Minimum window is "BANC".

/// Note:

/// If there is no such window in S that covers all characters in T,

/// return the empty string "".

/// If there are multiple such windows, you are guaranteed that there

/// will always be only one unique minimum window in S.

/// </summary>

string LeetCodeString::minWindow(string s, string t)

{

unordered\_map<char, int> source, target;

for (size\_t i = 0; i < t.size(); i++)

{

target[t[i]]++;

}

int left = -1;

string result;

int count = 0;

for (int right = 0; right < (int)s.size(); right++)

{

if (target.count(s[right]) == 0)

{

continue;

}

// not interested

source[s[right]]++;

if (source[s[right]] == target[s[right]])

{

count++;

}

while (count == target.size())

{

if (result.empty() || right - left < (int)result.size())

{

result = s.substr(left + 1, right - left);

}

left++;

// not interested

if (target.count(s[left]) == 0)

{

continue;

}

source[s[left]]--;

if (source[s[left]] == target[s[left]] - 1)

{

count--;

}

}

}

return result;

}