LeetCode Training Day 4 Two Pointers

Two-pointers is a common method when iterate an array or substring. You can put two pointers at each side of array and move to middle, therefore you reduce the scope. You can also put two pointers as greedy pattern, say first move the first pointer towards the end, increase the scope of search, then move the second pointer towards the first point to get minimum scope which satisfies the requirements.

## 125. Valid Palindrome

Easy

A phrase is a **palindrome** if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string s, return true*if it is a****palindrome****, or*false*otherwise*.

**Example 1:**

**Input:** s = "A man, a plan, a canal: Panama"

**Output:** true

**Explanation:** "amanaplanacanalpanama" is a palindrome.

**Example 2:**

**Input:** s = "race a car"

**Output:** false

**Explanation:** "raceacar" is not a palindrome.

**Example 3:**

**Input:** s = " "

**Output:** true

**Explanation:** s is an empty string "" after removing non-alphanumeric characters.

Since an empty string reads the same forward and backward, it is a palindrome.

**Constraints:**

* 1 <= s.length <= 2 \* 105
* s consists only of printable ASCII characters.

### Analysis:

We iterate the string from beginning and end. Move the two pointers from two ends to center, skip the non alphabet and convert the alphabet to lower case and check if they match. If any two characters do not match return false. If we complete iteration and can not find any mismatch return true.

/// <summary>

/// Leet code #125. Valid Palindrome

///

/// Given a string, determine if it is a palindrome, considering only

/// alphanumeric characters and ignoring cases.

/// For example,

/// "A man, a plan, a canal: Panama" is a palindrome.

/// "race a car" is not a palindrome.

/// Notes:

/// Have you consider that the string might be empty? This is a good

/// question to ask during an interview.

/// For the purpose of this problem, we define empty string as valid

/// palindrome.

/// </summary>

bool LeetCodeString::isPalindrome(string s)

{

int first = 0;

int last = s.size() - 1;

while (first < last)

{

if (!isalnum(s[first]))

{

first++;

}

else if (!isalnum(s[last]))

{

last--;

}

else if (tolower(s[first]) == tolower(s[last]))

{

first++;

last--;

}

else

{

return false;

}

}

return true;

}

## 680. Valid Palindrome II

Easy

Given a string s, return true *if the*s*can be palindrome after deleting****at most one****character from it*.

**Example 1:**

**Input:** s = "aba"

**Output:** true

**Example 2:**

**Input:** s = "abca"

**Output:** true

**Explanation:** You could delete the character 'c'.

**Example 3:**

**Input:** s = "abc"

**Output:** false

**Constraints:**

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

### Analysis:

We iterate the string from beginning and end by two pointers. If the two letters match each other, we keep on move two pointers closer until they meet, when meet, we return true. If the two letters do not match, we will move either left or right one step futher by deleting that letter and try again, if we found another mismatch, or say we used by the power of deleting characters, then we drop the current process. If we can not find a match palindrome, we return false.

The whole process can be done in a recursive way or use stack. The code is good for deleting at most N letters as well.

/// <summary>

/// Leet Code 680. Valid Palindrome II

///

/// Easy

///

/// Given a string s, return true if the s can be palindrome after

/// deleting at most one character from it.

///

/// Example 1:

/// Input: s = "aba"

/// Output: true

///

/// Example 2:

/// Input: s = "abca"

/// Output: true

/// Explanation: You could delete the character 'c'.

///

/// Example 3:

/// Input: s = "abc"

/// Output: false

///

/// Constraints:

/// 1. 1 <= s.length <= 10^5

/// 2. s consists of lowercase English letters.

/// </summary>

bool LeetCodeTwoPointer::validPalindrome(string s)

{

stack<vector<int>> stk;

stk.push({ 0, (int)s.size() - 1, 1 });

while (!stk.empty())

{

vector<int> pos = stk.top();

stk.pop();

while (s[pos[0]] == s[pos[1]] && pos[0] < pos[1])

{

pos[0]++;

pos[1]--;

}

if (pos[0] >= pos[1])

{

return true;

}

if (pos[2] == 0) // we already delete one letter

{

continue;

}

else

{

pos[2]--;

stk.push({ pos[0] + 1, pos[1], pos[2] });

stk.push({ pos[0], pos[1] - 1, pos[2] });

}

}

return false;

}

## 658. Find K Closest Elements

Medium

Given a **sorted** integer array arr, two integers k and x, return the k closest integers to x in the array. The result should also be sorted in ascending order.

An integer a is closer to x than an integer b if:

* |a - x| < |b - x|, or
* |a - x| == |b - x| and a < b

**Example 1:**

**Input:** arr = [1,2,3,4,5], k = 4, x = 3

**Output:** [1,2,3,4]

**Example 2:**

**Input:** arr = [1,2,3,4,5], k = 4, x = -1

**Output:** [1,2,3,4]

**Constraints:**

* 1 <= k <= arr.length
* 1 <= arr.length <= 104
* arr is sorted in **ascending** order.
* -104 <= arr[i], x <= 104

### Analysis:

We iterate from both ends and check which one has a closer distance.

/// <summary>

/// Leet Code 658. Find K Closest Elements

///

/// Medium

///

/// Given a sorted integer array arr, two integers k and x, return the k

/// closest integers to x in the array. The result should also be sorted

/// in ascending order.

///

/// An integer a is closer to x than an integer b if:

///

/// |a - x| < |b - x|, or

/// |a - x| == |b - x| and a < b

///

/// Example 1:

/// Input: arr = [1,2,3,4,5], k = 4, x = 3

/// Output: [1,2,3,4]

///

/// Example 2:

/// Input: arr = [1,2,3,4,5], k = 4, x = -1

/// Output: [1,2,3,4]

///

/// Constraints:

/// 1. 1 <= k <= arr.length

/// 2. 1 <= arr.length <= 10^4

/// 3. arr is sorted in ascending order.

/// 4. -10^4 <= arr[i], x <= 10^4

/// </summary>

vector<int> LeetCodeTwoPointer::findClosestElements(vector<int>& arr, int k, int x)

{

int first = 0;

int last = arr.size() - 1;

while (last - first >= k)

{

if (abs(arr[first] - x) > abs(arr[last] - x))

{

first++;

}

else

{

last--;

}

}

vector<int> result;

for (int i = first; i <= last; i++)

{

result.push\_back(arr[i]);

}

return result;

}

## 922. Sort Array By Parity II

Easy

Given an array of integers nums, half of the integers in nums are **odd**, and the other half are **even**.

Sort the array so that whenever nums[i] is odd, i is **odd**, and whenever nums[i] is even, i is **even**.

Return *any answer array that satisfies this condition*.

**Example 1:**

**Input:** nums = [4,2,5,7]

**Output:** [4,5,2,7]

**Explanation:** [4,7,2,5], [2,5,4,7], [2,7,4,5] would also have been accepted.

**Example 2:**

**Input:** nums = [2,3]

**Output:** [2,3]

**Constraints:**

* 2 <= nums.length <= 2 \* 104
* nums.length is even.
* Half of the integers in nums are even.
* 0 <= nums[i] <= 1000

### Analysis:

Keep the two pointers one pointing to even position and another pointing to odd position, move them if number match and swap them when both mismatches

/// <summary>

/// Leet code #922. Sort Array By Parity II

///

/// Given an array A of non-negative integers, half of the integers in A are

/// odd, and half of the integers are even.

///

/// Sort the array so that whenever A[i] is odd, i is odd; and whenever A[i]

/// is even, i is even.

///

/// You may return any answer array that satisfies this condition.

///

/// Example 1:

///

/// Input: [4,2,5,7]

/// Output: [4,5,2,7]

/// Explanation: [4,7,2,5], [2,5,4,7], [2,7,4,5] would also have been accepted.

///

/// Note:

/// 1. 2 <= A.length <= 20000

/// 2. A.length % 2 == 0

/// 3. 0 <= A[i] <= 1000

/// </summary>

vector<int> LeetCodeTwoPointer::sortArrayByParityII(vector<int>& A)

{

size\_t even = 0;

size\_t odd = 1;

vector<int> result = A;

while (even < A.size())

{

if (result[even] % 2 == 0) even += 2;

else

{

while ((odd < A.size()) && (result[odd] % 2 == 1)) odd += 2;

std::swap(result[even], result[odd]);

}

}

return result;

}

## 881. Boats to Save People

Medium

You are given an array people where people[i] is the weight of the ith person, and an **infinite number of boats** where each boat can carry a maximum weight of limit. Each boat carries at most two people at the same time, provided the sum of the weight of those people is at most limit.

Return *the minimum number of boats to carry every given person*.

**Example 1:**

**Input:** people = [1,2], limit = 3

**Output:** 1

**Explanation:** 1 boat (1, 2)

**Example 2:**

**Input:** people = [3,2,2,1], limit = 3

**Output:** 3

**Explanation:** 3 boats (1, 2), (2) and (3)

**Example 3:**

**Input:** people = [3,5,3,4], limit = 5

**Output:** 4

**Explanation:** 4 boats (3), (3), (4), (5)

**Constraints:**

* 1 <= people.length <= 5 \* 104
* 1 <= people[i] <= limit <= 3 \* 104

### Analysis:

Match the heaviest with lightest.

/// <summary>

/// Leet code #881. Boats to Save People

///

/// The i-th person has weight people[i], and each boat can carry a maximum

/// weight of limit.

///

/// Each boat carries at most 2 people at the same time, provided the sum of

/// the weight of those people is at most limit.

///

/// Return the minimum number of boats to carry every given person. (It is

/// guaranteed each person can be carried by a boat.)

///

/// Example 1:

///

/// Input: people = [1,2], limit = 3

/// Output: 1

/// Explanation: 1 boat (1, 2)

///

/// Example 2:

///

/// Input: people = [3,2,2,1], limit = 3

/// Output: 3

/// Explanation: 3 boats (1, 2), (2) and (3)

///

/// Example 3:

///

/// Input: people = [3,5,3,4], limit = 5

/// Output: 4

/// Explanation: 4 boats (3), (3), (4), (5)

/// Note:

///

/// 1. 1 <= people.length <= 50000

/// 2. 1 <= people[i] <= limit <= 30000

/// </summary>

int LeetCodeTwoPointer::numRescueBoats(vector<int>& people, int limit)

{

int result = 0;

sort(people.begin(), people.end());

int first = 0;

int last = people.size() - 1;

while (first <= last)

{

if (people[first] + people[last] <= limit)

{

first++;

}

last--;

result++;

}

return result;

}

## 1100. Find K-Length Substrings With No Repeated Characters

Medium

Given a string s and an integer k, return *the number of substrings in*s*of length*k*with no repeated characters*.

**Example 1:**

**Input:** s = "havefunonleetcode", k = 5

**Output:** 6

**Explanation:** There are 6 substrings they are: 'havef','avefu','vefun','efuno','etcod','tcode'.

**Example 2:**

**Input:** s = "home", k = 5

**Output:** 0

**Explanation:** Notice k can be larger than the length of s. In this case, it is not possible to find any substring.

**Constraints:**

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

### Analysis:

Keep the two pointers with K distance, track the charaters count in array and make sure none of the character with count more than 1.

/// <summary>

/// Leet code #1100. Find K-Length Substrings With No Repeated Characters

///

/// Given a string S, return the number of substrings of length K with no

/// repeated characters.

///

/// Example 1:

/// Input: S = "havefunonleetcode", K = 5

/// Output: 6

/// Explanation:

/// There are 6 substrings they are : 'havef','avefu','vefun','efuno',

/// 'etcod','tcode'.

///

/// Example 2:

/// Input: S = "home", K = 5

/// Output: 0

/// Explanation:

/// Notice K can be larger than the length of S. In this case is not

/// possible to find any substring.

///

/// Note:

/// 1 <= S.length <= 10^4

/// 2. All characters of S are lowercase English letters.

/// 3. 1 <= K <= 10^4

/// </summary>

int LeetCodeTwoPointer::numKLenSubstrNoRepeats(string S, int K)

{

vector<int> substring(26);

int first = 0;

int last = 0;

int result = 0;

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

{

if (substring[S[last] - 'a'] == 1)

{

substring[S[first] - 'a'] = 0;

first++;

}

else

{

substring[S[last] - 'a'] = 1;

last++;

if (last - first == K)

{

result++;

substring[S[first] - 'a'] = 0;

first++;

}

}

}

return result;

}

## 11. Container With Most Water

Medium

Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of the line i is at (i, ai) and (i, 0). Find two lines, which, together with the x-axis forms a container, such that the container contains the most water.

**Notice** that you may not slant the container.

**Example 1:**

A picture containing bar chart

Description automatically generated

**Input:** height = [1,8,6,2,5,4,8,3,7]

**Output:** 49

**Explanation:** The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

**Example 2:**

**Input:** height = [1,1]

**Output:** 1

**Example 3:**

**Input:** height = [4,3,2,1,4]

**Output:** 16

**Example 4:**

**Input:** height = [1,2,1]

**Output:** 2

**Constraints:**

* n == height.length
* 2 <= n <= 105
* 0 <= height[i] <= 104

### Analysis:

Keep on moving the pthe two pointers with left and right, if left height is lower, move left pointer to right, if right pointer is lower move right pointer. Calculate the area and get maximum one.

/// <summary>

/// Leet Code 11. Container With Most Water

///

/// Medium

///

/// You are given an integer array height of length n. There are n

/// vertical lines drawn such that the two endpoints of the ith

/// line are (i, 0) and (i, height[i]).

///

/// Find two lines that together with the x-axis form a container,

/// such that the container contains the most water.

///

/// Return the maximum amount of water a container can store.

/// Notice that you may not slant the container.

///

/// Example 1:

/// Input: height = [1,8,6,2,5,4,8,3,7]

/// Output: 49

/// Explanation: The above vertical lines are represented by array

/// [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue

/// section) the container can contain is 49.

///

/// Example 2:

/// Input: height = [1,1]

/// Output: 1

///

/// Constraints:

/// 1. n == height.length

/// 2. 2 <= n <= 10^5

/// 3. 0 <= height[i] <= 10^4

/// </summary>

int LeetCodeTwoPointer::maxArea(vector<int>& height)

{

int maxArea = 0;

int first = 0, last = height.size() - 1;

while (first < last)

{

int area = min(height[first], height[last]) \* (last - first);

if (area > maxArea)

{

maxArea = area;

}

if (height[first] < height[last])

{

first++;

}

else

{

last--;

}

}

return maxArea;

}

## 1852. Distinct Numbers in Each Subarray

Medium

Given an integer array nums and an integer k, you are asked to construct the array ans of size n-k+1 where ans[i] is the number of **distinct** numbers in the subarray nums[i:i+k-1] = [nums[i], nums[i+1], ..., nums[i+k-1]].

Return *the array*ans.

**Example 1:**

**Input:** nums = [1,2,3,2,2,1,3], k = 3

**Output:** [3,2,2,2,3]

**Explanation:** The number of distinct elements in each subarray goes as follows:

- nums[0:2] = [1,2,3] so ans[0] = 3

- nums[1:3] = [2,3,2] so ans[1] = 2

- nums[2:4] = [3,2,2] so ans[2] = 2

- nums[3:5] = [2,2,1] so ans[3] = 2

- nums[4:6] = [2,1,3] so ans[4] = 3

**Example 2:**

**Input:** nums = [1,1,1,1,2,3,4], k = 4

**Output:** [1,2,3,4]

**Explanation:** The number of distinct elements in each subarray goes as follows:

- nums[0:3] = [1,1,1,1] so ans[0] = 1

- nums[1:4] = [1,1,1,2] so ans[1] = 2

- nums[2:5] = [1,1,2,3] so ans[2] = 3

- nums[3:6] = [1,2,3,4] so ans[3] = 4

**Constraints:**

* 1 <= k <= nums.length <= 105
* 1 <= nums[i] <= 105

### Analysis:

Maintain a k size slide window and count distintinct integers.

/// <summary>

/// Leet Code 1852. Distinct Numbers in Each Subarray

///

/// Medium

///

/// Given an integer array nums and an integer k, you are asked to

/// construct the array ans of size n-k+1 where ans[i] is the number

/// of distinct numbers in the subarray nums[i:i+k-1] = [nums[i],

/// nums[i+1], ..., nums[i+k-1]].

///

/// Return the array ans.

///

/// Example 1:

/// Input: nums = [1,2,3,2,2,1,3], k = 3

/// Output: [3,2,2,2,3]

/// Explanation: The number of distinct elements in each subarray goes

/// as follows:

/// - nums[0:2] = [1,2,3] so ans[0] = 3

/// - nums[1:3] = [2,3,2] so ans[1] = 2

/// - nums[2:4] = [3,2,2] so ans[2] = 2

/// - nums[3:5] = [2,2,1] so ans[3] = 2

/// - nums[4:6] = [2,1,3] so ans[4] = 3

///

/// Example 2:

/// Input: nums = [1,1,1,1,2,3,4], k = 4

/// Output: [1,2,3,4]

/// Explanation: The number of distinct elements in each subarray goes

/// as follows:

/// - nums[0:3] = [1,1,1,1] so ans[0] = 1

/// - nums[1:4] = [1,1,1,2] so ans[1] = 2

/// - nums[2:5] = [1,1,2,3] so ans[2] = 3

/// - nums[3:6] = [1,2,3,4] so ans[3] = 4

///

/// Constraints:

/// 1. 1 <= k <= nums.length <= 10^5

/// 2. 1 <= nums[i] <= 10^5

/// </summary>

vector<int> LeetCodeTwoPointer::distinctNumbers(vector<int>& nums, int k)

{

unordered\_map<int, int> num\_map;

int first = 0;

int last = 0;

vector<int> result;

while (last < k - 1)

{

num\_map[nums[last]]++;

last++;

}

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

{

num\_map[nums[last]]++;

result.push\_back(num\_map.size());

num\_map[nums[first]]--;

if (num\_map[nums[first]] == 0)

{

num\_map.erase(nums[first]);

}

first++;

last++;

}

return result;

}

## 1004. Max Consecutive Ones III

Medium

Given an array A of 0s and 1s, we may change up to K values from 0 to 1.

Return the length of the longest (contiguous) subarray that contains only 1s.

**Example 1:**

**Input:** A = [1,1,1,0,0,0,1,1,1,1,0], K = 2

**Output:** 6

**Explanation:**

[1,1,1,0,0,**1**,1,1,1,1,**1**]

Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.

**Example 2:**

**Input:** A = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], K = 3

**Output:** 10

**Explanation:**

[0,0,1,1,**1**,**1**,1,1,1,**1**,1,1,0,0,0,1,1,1,1]

Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.

**Note:**

1. 1 <= A.length <= 20000
2. 0 <= K <= A.length
3. A[i] is 0 or 1

### Analysis:

We can use two pointers to track the subarray to keep the maximum K ‘0’ in the range. Before the number of ‘0’ goes beyond the scope, we can greedily expand the last pointer.

/// <summary>

/// Leet Code 1004. Max Consecutive Ones III

///

/// Medium

///

/// Given a binary array nums and an integer k, return the maximum number of

/// consecutive 1's in the array if you can flip at most k 0's.

///

/// Example 1:

/// Input: nums = [1,1,1,0,0,0,1,1,1,1,0], k = 2

/// Output: 6

/// Explanation: [1,1,1,0,0,1,1,1,1,1,1]

/// Bolded numbers were flipped from 0 to 1. The longest subarray is

/// underlined.

///

/// Example 2:

/// Input: nums = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], k = 3

/// Output: 10

/// Explanation: [0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1]

/// Bolded numbers were flipped from 0 to 1. The longest subarray is

/// underlined.

///

/// Constraints:

/// 1. 1 <= nums.length <= 10^5

/// 2. nums[i] is either 0 or 1.

/// 3. 0 <= k <= nums.length

/// </summary>

int LeetCodeTwoPointer::longestOnes(vector<int>& nums, int k)

{

int result = 0;

int prev = -1;

int zero = 0;

for (int i = 0; i < (int)nums.size(); i++)

{

if (nums[i] == 0) zero++;

while (zero > k)

{

prev++;

if (nums[prev] == 0) zero--;

}

result = max(result, i - prev);

}

return result;

}

## 2062. Count Vowel Substrings of a String

Easy

A **substring** is a contiguous (non-empty) sequence of characters within a string.

A **vowel substring** is a substring that **only** consists of vowels ('a', 'e', 'i', 'o', and 'u') and has **all five** vowels present in it.

Given a string word, return *the number of****vowel substrings****in* word.

**Example 1:**

**Input:** word = "aeiouu"

**Output:** 2

**Explanation:** The vowel substrings of word are as follows (underlined):

- "**aeiou**u"

- "**aeiouu**"

**Example 2:**

**Input:** word = "unicornarihan"

**Output:** 0

**Explanation:** Not all 5 vowels are present, so there are no vowel substrings.

**Example 3:**

**Input:** word = "cuaieuouac"

**Output:** 7

**Explanation:** The vowel substrings of word are as follows (underlined):

- "c**uaieuo**uac"

- "c**uaieuou**ac"

- "c**uaieuoua**c"

- "cu**aieuo**uac"

- "cu**aieuou**ac"

- "cu**aieuoua**c"

- "cua**ieuoua**c"

**Example 4:**

**Input:** word = "bbaeixoubb"

**Output:** 0

**Explanation:** The only substrings that contain all five vowels also contain consonants, so there are no vowel substrings.

**Constraints:**

* 1 <= word.length <= 100
* word consists of lowercase English letters only.

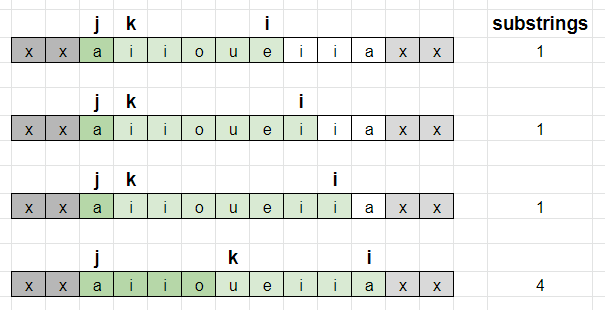
### Analysis:

Our target is to do it in O(N) time complexity, so it becomes a medium problem. This is a typical string coverage problem which can be resolved by two pointers with a concept of slide window. First, we scan on every index as end of substring, say using pointer of last. When we see an non-vowel we set first as -1 to say the current slide window is not valid. When we see a vowel, we can start to set up slide window if first is -1. On every vowel, we add the count, when a count from 0 to 1 we know a new vowel found, when we have 5 vowels, we know it is a valid substring.

We also want to know the minimum size of the slide window, so we use the third pointer, head. Head can start from pointer first, but can move forward as long as we do not invalidate the all vowels conditition, the distance between head and first is the number of substring we want to add in every scan point.

j mark the start of an "all-vowel" substring, and i is the current position. The window between k - 1 and i is the smallest window with all 5 vowels.

So, for each position i, we have k - j valid substrings. The picture below demonstrate it for "xxaiioueiiaxx" test case:



/// <summary>

/// Leet Code 2062. Count Vowel Substrings of a String

///

/// Easy

///

/// A substring is a contiguous (non-empty) sequence of characters within

/// a string.

///

/// A vowel substring is a substring that only consists of vowels

/// ('a', 'e', 'i', 'o', and 'u') and has all five vowels present in it.

/// Given a string word, return the number of vowel substrings in word.

///

/// Example 1:

/// Input: word = "aeiouu"

/// Output: 2

/// Explanation: The vowel substrings of word are as follows (underlined):

/// - "aeiouu"

/// - "aeiouu"

///

/// Example 2:

/// Input: word = "unicornarihan"

/// Output: 0

/// Explanation: Not all 5 vowels are present, so there are no vowel

/// substrings.

///

/// Example 3:

/// Input: word = "cuaieuouac"

/// Output: 7

/// Explanation: The vowel substrings of word are as follows (underlined):

/// - "cuaieuouac"

/// - "cuaieuouac"

/// - "cuaieuouac"

/// - "cuaieuouac"

/// - "cuaieuouac"

/// - "cuaieuouac"

/// - "cuaieuouac"

///

/// Example 4:

/// Input: word = "bbaeixoubb"

/// Output: 0

/// Explanation: The only substrings that contain all five vowels also

/// contain

/// consonants, so there are no vowel substrings.

///

/// Constraints:

/// 1. 1 <= word.length <= 100

/// 2. word consists of lowercase English letters only.

/// </summary>

int LeetCodeTwoPointer::countVowelSubstrings(string word)

{

int first = -1;

int last = 0;

int count = 0;

int head = 0;

int result = 0;

unordered\_map<char, int> vowel\_map =

{

{'a', 0}, {'e', 0}, {'i', 0}, {'o', 0}, {'u', 0}

};

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

{

if (vowel\_map.count(word[last]) > 0)

{

if (first == -1)

{

vowel\_map['a'] = vowel\_map['e'] = vowel\_map['i'] = vowel\_map['o'] = vowel\_map['u'] = 0;

first = last;

head = first;

count = 0;

}

vowel\_map[word[last]]++;

if (vowel\_map[word[last]] == 1) count++;

if (count == 5)

{

while(vowel\_map[word[head]] > 1)

{

vowel\_map[word[head]]--;

head++;

}

result += head - first + 1;

}

}

else

{

first = -1;

}

last++;

}

return result;

}

# Advance Problems

## 1156. Swap For Longest Repeated Character Substring

Medium

You are given a string text. You can swap two of the characters in the text.

Return *the length of the longest substring with repeated characters*.

**Example 1:**

**Input:** text = "ababa"

**Output:** 3

**Explanation:** We can swap the first 'b' with the last 'a', or the last 'b' with the first 'a'. Then, the longest repeated character substring is "aaa", which its length is 3.

**Example 2:**

**Input:** text = "aaabaaa"

**Output:** 6

**Explanation:** Swap 'b' with the last 'a' (or the first 'a'), and we get longest repeated character substring "aaaaaa", which its length is 6.

**Example 3:**

**Input:** text = "aaabbaaa"

**Output:** 4

**Example 4:**

**Input:** text = "aaaaa"

**Output:** 5

**Explanation:** No need to swap, longest repeated character substring is "aaaaa", length is 5.

**Example 5:**

**Input:** text = "abcdef"

**Output:** 1

**Constraints:**

1. 1 <= text.length <= 2 \* 104
2. text consist of lowercase English characters only.

### Analysis:

1. Keep two pointers moving, last pointer to expand greedly and first pointer to shrink if no such condition satisfied
2. For each character from ‘a’ to ‘z’ the condition to make substring all as this character is the number of non-such characters within the window equals to the number of such characters out of the window, so you need to calculate the count of each character in advance.

/// <summary>

/// Leet Code 1156. Swap For Longest Repeated Character Substring

///

/// Medium

///

/// You are given a string text. You can swap two of the characters in the

/// text.

///

/// Return the length of the longest substring with repeated characters.

///

/// Example 1:

/// Input: text = "ababa"

/// Output: 3

/// Explanation: We can swap the first 'b' with the last 'a', or the last

/// 'b' with the first 'a'. Then, the longest repeated character

/// substring is "aaa" with length 3.

///

/// Example 2:

/// Input: text = "aaabaaa"

/// Output: 6

/// Explanation: Swap 'b' with the last 'a' (or the first 'a'), and we get

/// longest repeated character substring "aaaaaa" with length 6.

///

/// Example 3:

/// Input: text = "aaaaa"

/// Output: 5

/// Explanation: No need to swap, longest repeated character substring is

/// "aaaaa" with length is 5.

///

/// Constraints:

/// 1. 1 <= text.length <= 2 \* 10^4

/// 2. text consist of lowercase English characters only.

/// </summary>

int LeetCodeTwoPointer::maxRepOpt1(string text)

{

vector<int> count(26);

vector<int> window(26);

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

{

count[text[i] - 'a']++;

}

int first = 0;

int last = 0;

int result = 0;

window[text[0] - 'a']++;

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

{

bool match = false;

int length = last - first + 1;

for (int i = 0; i < 26; i++)

{

if (window[i] == length)

{

match = true;

result = max(result, length);

break;

}

else if ((window[i] == length - 1) && (count[i] > window[i]))

{

match = true;

result = max(result, length);

break;

}

}

if (match)

{

last++;

if (last < (int)text.size()) window[text[last] - 'a']++;

}

else

{

window[text[first] - 'a']--;

first++;

}

}

return result;

}

## 42. Trapping Rain Water

Hard

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

**Example 1:**

A picture containing text, clipart

Description automatically generated

**Input:** height = [0,1,0,2,1,0,1,3,2,1,2,1]

**Output:** 6

**Explanation:** The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

**Example 2:**

**Input:** height = [4,2,0,3,2,5]

**Output:** 9

**Constraints:**

* n == height.length
* 1 <= n <= 2 \* 104
* 0 <= height[i] <= 105

### Analysis:

Keep the two pointers with left and right, if left height is lower, move left pointer to right, if right pointer is lower move right pointer.

/// <summary>

/// Leet Code 42. Trapping Rain Water

///

/// Hard

///

/// Given n non-negative integers representing an elevation map where the

/// width of each bar is 1, compute how much water it can trap after

/// raining.

///

/// Example 1:

/// Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]

/// Output: 6

/// Explanation: The above elevation map (black section) is represented

/// by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain

/// water (blue section) are being trapped.

///

/// Example 2:

/// Input: height = [4,2,0,3,2,5]

/// Output: 9

///

/// Constraints:

/// 1. n == height.length

/// 2. 1 <= n <= 2 \* 10^4

/// 3. 0 <= height[i] <= 10^5

/// </summary>

int LeetCodeTwoPointer::trap(vector<int>& height)

{

int sum = 0;

int left\_index = 0;

int right\_index = height.size() - 1;

int left\_value = height[left\_index];

int right\_value = height[right\_index];

while (left\_index < right\_index)

{

if (left\_value < right\_value)

{

left\_index++;

// if left side become lower, take the water and fill it

if (height[left\_index] < left\_value)

{

sum = sum + left\_value - height[left\_index];

}

else

{

left\_value = height[left\_index];

}

}

else

{

right\_index--;

// if right side become lower, take the water and fill it

if (height[right\_index] < right\_value)

{

sum = sum + right\_value - height[right\_index];

}

else

{

right\_value = height[right\_index];

}

}

}

return sum;

}