# Word Coverage

The simplest pattern in string problem is Word Coverage. The basic pattern is that given a word, and a string, find a minimum substring which cover all the characters in the word.

First you count the characters in the word in an array, then you scan the string with a slide window, for any character hit the word, deduct the count, when all the character count in the array reduced to zero we have a full coverage for the word.

## 383. Ransom Note

Easy

Given an arbitrary ransom note string and another string containing letters from all the magazines, write a function that will return true if the ransom note can be constructed from the magazines ; otherwise, it will return false.

Each letter in the magazine string can only be used once in your ransom note.

**Note:**  
You may assume that both strings contain only lowercase letters.

canConstruct("a", "b") -> false

canConstruct("aa", "ab") -> false

canConstruct("aa", "aab") -> true

### **Analysis**

**Use hash table to count characters in magazine and deduct the character in ransom note.**

/// <summary>

/// Leet code #383. Ransom Note

///

/// Given an arbitrary ransom note string and another string containing

/// letters from all the magazines, write a function that will return true

/// if the ransom note can be constructed from the magazines ; otherwise,

/// it will return false.

///

/// Each letter in the magazine string can only be used once in your ransom note.

/// Note:

/// You may assume that both strings contain only lowercase letters.

/// canConstruct("a", "b") -> false

/// canConstruct("aa", "ab") -> false

/// canConstruct("aa", "aab") -> true

/// </summary>

bool LeetCodeString::canConstruct(string ransomNote, string magazine)

{

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

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

{

char\_map[magazine[i]]++;

}

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

{

char\_map[ransomNote[i]]--;

if (char\_map[ransomNote[i]] < 0) return false;

}

return true;

}

## 3. Longest Substring Without Repeating Characters

Medium

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

**Example 1:**

**Input:** "abcabcbb"

**Output:** 3

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

**Example 2:**

**Input:** "bbbbb"

**Output:** 1

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

**Example 3:**

**Input:** "pwwkew"

**Output:** 3

**Explanation:** 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.

### **Analysis**

**Use hash table to count characters and use slide window to get maximum substring.**

/// <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 = 0;

int result = 0;

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

{

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

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

{

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

begin++;

}

result = max(result, end - begin + 1);

}

return result;

}

## 76. Minimum Window Substring

Hard

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).

**Example:**

**Input: S** = "ADOBECODEBANC", **T** = "ABC"

**Output:** "BANC"

**Note:**

* If there is no such window in S that covers all characters in T, return the empty string "".
* If there is such window, you are guaranteed that there will always be only one unique minimum window in S.

### **Analysis**

**Have a slide window, which count the character coverage for the word T, keep on moving last pointer until fully covered, the move first pointer to make minimum length of substring. During the scan we may have negative count for some characters which should not be considered as a valid coverage count.**

/// <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)

{

vector<int> char\_map(128);

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

{

char\_map[t[i]]++;

}

int count = t.size();

int begin = 0;

string result;

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

{

if (char\_map[s[end]] > 0)

{

count--;

}

char\_map[s[end]]--;

// Do we have all the characters matched

if (count > 0) continue;

// reduce slide window, until break the condition

while (count == 0)

{

if (result.empty() || (end - begin + 1) < (int)result.size())

{

result = s.substr(begin, end - begin + 1);

}

char\_map[s[begin]]++;

if (char\_map[s[begin]] > 0)

{

count++;

}

begin++;

}

}

return result;

}

## 423. Reconstruct Original Digits from English

Medium

Given a **non-empty** string containing an out-of-order English representation of digits 0-9, output the digits in ascending order.

**Note:**

1. Input contains only lowercase English letters.
2. Input is guaranteed to be valid and can be transformed to its original digits. That means invalid inputs such as "abc" or "zerone" are not permitted.
3. Input length is less than 50,000.

**Example 1:**

Input: "owoztneoer"

Output: "012"

**Example 2:**

Input: "fviefuro"

Output: "45"

### **Analysis**

**This is more like an IQ test, we have ‘z’ only for zero, ‘w’ only for two, ‘u’ for four, ‘x’ for six, … We can conclude until every digits.**

/// <summary>

/// Leet code #423. Reconstruct Original Digits from English

///

/// Given a non-empty string containing an out-of-order English representation

/// of digits 0-9, output the digits in ascending order.

/// Note:

/// 1.Input contains only lowercase English letters.

/// 2.Input is guaranteed to be valid and can be transformed to its original

/// digits.That means invalid inputs such as "abc" or "zerone" are not

/// permitted.

/// 3.Input length is less than 50,000.

/// Example 1:

/// Input: "owoztneoer"

/// Output: "012"

/// Input: "fviefuro"

/// Example 2:

/// Output: "45"

/// </summary>

string LeetCodeString::originalDigits(string s)

{

string result;

vector<string> digit\_map =

{

"zero", "one", "two", "three", "four", "five", "six", "seven",

"eight", "nine"

};

vector<int> digits(10);

vector<int> alphabet\_count(26);

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

{

alphabet\_count[s[i] - 'a'] ++;

}

digits[0] = alphabet\_count['z' - 'a'];

digits[2] = alphabet\_count['w' - 'a'];

digits[4] = alphabet\_count['u' - 'a'];

digits[6] = alphabet\_count['x' - 'a'];

digits[8] = alphabet\_count['g' - 'a'];

digits[7] = alphabet\_count['s' - 'a'] - digits[6];

digits[3] = alphabet\_count['h' - 'a'] - digits[8];

digits[5] = alphabet\_count['f' - 'a'] - digits[4];

digits[1] = alphabet\_count['o' - 'a'] - digits[0] - digits[2] - digits[4];

digits[9] = alphabet\_count['i' - 'a'] - digits[5] - digits[6] - digits[8];

for (size\_t i = 0; i < 10; i++)

{

result.append(string(digits[i], (char)('0' + i)));

}

return result;

}

## 438. Find All Anagrams in a String

Medium

Given a string **s** and a **non-empty** string **p**, find all the start indices of **p**'s anagrams in **s**.

Strings consists of lowercase English letters only and the length of both strings **s** and **p** will not be larger than 20,100.

The order of output does not matter.

**Example 1:**

**Input:**

s: "cbaebabacd" p: "abc"

**Output:**

[0, 6]

**Explanation:**

The substring with start index = 0 is "cba", which is an anagram of "abc".

The substring with start index = 6 is "bac", which is an anagram of "abc".

**Example 2:**

**Input:**

s: "abab" p: "ab"

**Output:**

[0, 1, 2]

**Explanation:**

The substring with start index = 0 is "ab", which is an anagram of "ab".

The substring with start index = 1 is "ba", which is an anagram of "ab".

The substring with start index = 2 is "ab", which is an anagram of "ab".

### **Analysis**

**Keep the slide window size as length of p and count the characters.**

/// <summary>

/// Leet code #438. Find All Anagrams in a String

///

/// Given a string s and a non-empty string p, find all the start

/// indices of p's anagrams in s.

/// Strings consists of lowercase English letters only and the length

/// of both strings s and p will not be larger than 20,100.

/// The order of output does not matter.

///

/// Example 1:

/// Input:

/// s: "cbaebabacd" p: "abc"

/// Output:

/// [0, 6]

/// Explanation:

/// The substring with start index = 0 is "cba", which is an anagram of "abc".

/// The substring with start index = 6 is "bac", which is an anagram of "abc".

///

/// Example 2:

/// Input:

/// s: "abab" p: "ab"

/// Output:

/// [0, 1, 2]

/// Explanation:

/// The substring with start index = 0 is "ab", which is an anagram of "ab".

/// The substring with start index = 1 is "ba", which is an anagram of "ab".

/// The substring with start index = 2 is "ab", which is an anagram of "ab".

/// </summary>

vector<int> LeetCodeString::findAnagrams(string s, string p)

{

vector<int> result;

int char\_count = 0;

vector<int> char\_map(26);

for (char ch : p) char\_map[ch - 'a']++;

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

{

if (i >= p.size())

{

char\_map[s[i - p.size()] - 'a']++;

if (char\_map[s[i - p.size()] - 'a'] > 0) char\_count--;

}

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

if (char\_map[s[i] - 'a'] >= 0) char\_count++;

if (char\_count == p.size())

{

result.push\_back(i - char\_count + 1);

}

}

return result;

}

## 424. Longest Repeating Character Replacement

Medium

Given a string s that consists of only uppercase English letters, you can perform at most k operations on that string.

In one operation, you can choose **any** character of the string and change it to any other uppercase English character.

Find the length of the longest sub-string containing all repeating letters you can get after performing the above operations.

**Note:**  
Both the string's length and *k* will not exceed 104.

**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.

### **Analysis**

**The key to this problem is to track the maximum character count, if maximum character count + K is greater than or equal to the current length then current length is valid, will keep it as result, but if character count + K is less than the current length, then current length is invalid, we need to shrink the window until we found another greater max character count.**

**Please remember we do not care when we shrink the window, we remove the maximum count character or not.**

/// <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 104.

///

/// 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;

}