DSA Strings

LeetCode - String I

#	Problem Title	#	Problem Title
01	remove vowels from a string	02	defanging an ip address
03	goal parser interpretation	04	to lower case
05	replace all digits with characters	06	determine if string halves are alike
07	decrypt string from alphabet to integer mapping	08	generate a string with characters that have odd counts
09	number of strings that appear as substrings in word	10	robot return to origin
11	merge strings alternately	12	reverse words in a string iii
13	check if word equals summation of two words	14	reverse string

LeetCode - String II

#	Problem Title	#	Problem Title
01	remove palindromic subsequences	02	goat latin
03	reformat phone number	04	check if a word occurs as a prefix of any word in a sentence
05	string matching in an array	06	sum of digits of string after convert
07	reverse vowels of a string	08	rearrange spaces between words

09	repeated substring pattern	10	license key formatting
11	valid palindrome ii	12	maximum repeating substring
13	read n characters given read4	14	number of segments in a string

Codeforces - String

#	Problem Title	#	Problem Title
01	Round 148 B	02	Round 170 B
03	Round 173 B	04	Round 199 B
05	Round 208 B	06	Round 253 B
07	Round 336 B	08	Round 389 B
09	Round 47 B	10	Round 67 B
11	Round 442 B	12	Round 24 B
13	Round 73 B	14	Round 31 B
15	Round 106 B	16	Round 158 B
17	Round 335 B	18	Round 425 B
19	Round 64 B	20	Round 36 B
21	Round 417 B	22	Round 168 B
23	Round 114 B	24	Round 127 B
25	Round 48 B	26	Round 30 B
27	Round 307 B	28	Round 347 B

Commonly Used String Functions in C++

The std::string class contains functions to provide some common string operations. The below table contains some of the most commonly used functions in C++:

S. No.	**Category**	**Functions and Operators**	**Functionality**

1.	**String Length**	**length() or size()**	It will return the length of the string.
2.	**Accessing Characters**	**Indexing (using array[index])**	To access individual characters using array indexing.
		at()	Used to access a character at a specified index.
3.	**Appending and Concatenating Strings**	***+ Operator***	+ operator is used to concatenate two strings.
		append()	The append() function adds one string to the end of another.
4.	**String Comparison**	***== Operator***	You can compare strings using the == operator.
		compare()	The compare() function returns an integer value indicating the comparison result.
5.	**Substrings**	**substr()**	Use the substr() function to extract a substring from a string.
6.	**Searching**	**find()**	The find() function returns the position of the first occurrence of a substring.
7.	**Modifying Strings**	**replace()**	Use the replace() function to modify a part of the string.
		insert()	The insert() function adds a substring at a specified position.
		erase()	Use the erase() function to remove a part of the string.
8.	**Conversion**	**c_str()**	To obtain a C-style string from a std::string, you can use the c_str() function.

LeetCode - String II

1332. Remove Palindromic Subsequences

Easy

Topics

Companies

Hint

You are given a string `s` consisting **only** of letters `'a'` and `'b'`. In a single step you can remove one **palindromic subsequence** from `s`.

Return *the **minimum** number of steps to make the given string empty*.

A string is a **subsequence** of a given string if it is generated by deleting some characters of a given string without changing its order. Note that a subsequence does **not** necessarily need to be contiguous.

A string is called **palindrome** if is one that reads the same backward as well as forward.

Example 1:

```
Input: s = "ababa"
Output: 1
Explanation: s is already a palindrome, so its entirety can b
e removed in a single step.
```

Example 2:

```
Input: s = "abb"
Output: 2
Explanation: "abb" -> "bb" -> "".
Remove palindromic subsequence "a" then "bb".
```

Example 3:

```
Input: s = "baabb"
Output: 2
Explanation: "baabb" -> "b" -> "".
Remove palindromic subsequence "baab" then "b".
```

Constraints:

- `1 <= s.length <= 1000`
- `s[i]` is either `'a'` or `'b'`.

code :

```
class Solution {
public:
    int removePalindromeSub(string s) {
        int cnt = 0;

        int i= 0;
        int j=s.size()-1;
        while (i<j) {
            if (s[i] != s[j]) {
                return 2;
            }
            i++;
            j--;
        }
        return 1 ;



    }
};
```

824. Goat Latin

Easy

Topics

Companies

You are given a string `sentence` that consist of words separated by spaces. Each word consists of lowercase and uppercase letters only.

We would like to convert the sentence to "Goat Latin" (a made-up language similar to Pig Latin.) The rules of Goat Latin are as follows:

- If a word begins with a vowel (`'a'`, `'e'`, `'i'`, `'o'`, or `'u'`), append `"ma"` to the end of the word.
    - For example, the word `"apple"` becomes `"applema"`.
- If a word begins with a consonant (i.e., not a vowel), remove the first letter and append it to the end, then add `"ma"`.
    - For example, the word `"goat"` becomes `"oatgma"`.
- Add one letter `'a'` to the end of each word per its word index in the sentence, starting with `1`.
    - For example, the first word gets `"a"` added to the end, the second word gets `"aa"` added to the end, and so on.

Return *the final sentence representing the conversion from sentence to Goat Latin*.

Example 1:

```
Input: sentence = "I speak Goat Latin"
Output: "Imaa peaksmaaa oatGmaaaa atinLmaaaaa"
```

Example 2:

```
Input: sentence = "The quick brown fox jumped over the lazy d
og"
```

```
Output: "heTmaa uickqmaaa rownbmaaaa oxfmaaaaa umpedjmaaaaaa
overmaaaaaaa hetmaaaaaaaa azylmaaaaaaaaa ogdmaaaaaaaaaa"
```

Constraints:

- `1 <= sentence.length <= 150`
- `sentence` consists of English letters and spaces.
- `sentence` has no leading or trailing spaces.
- All the words in `sentence` are separated by a single space.

code :

```
class Solution {
public:
    string toGoatLatin(string sentence) {
        vector<string> words;
        stringstream ss(sentence);
        string tmp;
        int index = 1;

        while (getline(ss, tmp, ' ')) {
            char front = tolower(tmp.front());

            if (front == 'a' || front == 'e' || front == 'i' ||
                tmp.append("ma");
            } else {
                char firstLetter = tmp.front();
                tmp.erase(0, 1);
                tmp.push_back(firstLetter);
                tmp.append("ma");
            }

            tmp.append(string(index, 'a'));  // Add 'a' based o
            words.push_back(tmp);
            index++;
        }

        string result = "";
        for (int i = 0; i < words.size(); i++) {
            result += words[i];
            if (i != words.size() - 1) {
                result += " ";
            }
        }

        return result;
    }
};
```

Step-by-Step Explanation

1. **Splitting the Sentence into Words:**

   - We first split the input sentence into individual words. This is done using a `stringstream` and `getline` to extract words separated by spaces.

   ```
   cppCopy code
   while (getline(ss, tmp, ' ')) {
       // Each word is extracted here
   }
   ```

2. **Processing Each Word:**

   - For each word, we check if the first character is a vowel or a consonant.
   - **If the word starts with a vowel**: We directly append "ma" to the word.
   - **If the word starts with a consonant**: We remove the first letter, append it to the end of the word, and then add "ma".

```
cppCopy code
char front = tolower(tmp.front());

if (front == 'a' || front == 'e' || front == 'i' || front
== 'o' || front == 'u') {
    tmp.append("ma");
} else {
    char firstLetter = tmp.front();
    tmp.erase(0, 1);
    tmp.push_back(firstLetter);
    tmp.append("ma");
}
```

3. **Appending 'a' Based on Word Position:**
   - After the transformation, we append a sequence of 'a' characters. The number of 'a's corresponds to the position of the word in the sentence (1st word gets 1 'a', 2nd word gets 2 'a's, and so on).

```
cppCopy code
tmp.append(string(index, 'a'));
```

4. **Building the Final Result:**
   - The processed words are stored in a vector, and finally, we concatenate them into a single string separated by spaces.
   - We ensure there isn't an extra space at the end of the sentence.

```
cppCopy code
string result = "";
for (int i = 0; i < words.size(); i++) {
    result += words[i];
    if (i != words.size() - 1) {
        result += " ";
    }
}
```

5. **Returning the Result:**
    - The final result, which is the sentence in "Goat Latin," is returned.

Summary

- **Input**: A sentence with words separated by spaces.
- **Output**: The sentence transformed into "Goat Latin" following the specified rules.

The code ensures that each word is processed according to its initial character, and the appropriate suffix ("ma" + sequence of 'a's) is appended. The result is built without any extra spaces at the end.

complexity time : O(N)

space complexity : O(N)

1694. Reformat Phone Number

Easy (ask sir)

Topics

Companies

Hint

You are given a phone number as a string `number`. `number` consists of digits, spaces `' '`, and/or dashes `'-'`.

You would like to reformat the phone number in a certain manner.
Firstly, **remove** all spaces and dashes. Then, **group** the digits from left to right into blocks of length 3 **until** there are 4 or fewer digits. The final digits are then grouped as follows:

- 2 digits: A single block of length 2.

- 3 digits: A single block of length 3.
- 4 digits: Two blocks of length 2 each.

The blocks are then joined by dashes. Notice that the reformatting process should **never** produce any blocks of length 1 and produce **at most** two blocks of length 2.

Return *the phone number after formatting.*

Example 1:

```
Input: number = "1-23-45 6"
Output: "123-456"
Explanation: The digits are "123456".
Step 1: There are more than 4 digits, so group the next 3 dig
its. The 1st block is "123".
Step 2: There are 3 digits remaining, so put them in a single
block of length 3. The 2nd block is "456".
Joining the blocks gives "123-456".
```

Example 2:

```
Input: number = "123 4-567"
Output: "123-45-67"
Explanation:The digits are "1234567".
Step 1: There are more than 4 digits, so group the next 3 dig
its. The 1st block is "123".
Step 2: There are 4 digits left, so split them into two block
s of length 2. The blocks are "45" and "67".
Joining the blocks gives "123-45-67".
```

Example 3:

```
Input: number = "123 4-5678"
Output: "123-456-78"
Explanation: The digits are "12345678".
Step 1: The 1st block is "123".
Step 2: The 2nd block is "456".
Step 3: There are 2 digits left, so put them in a single bloc
k of length 2. The 3rd block is "78".
Joining the blocks gives "123-456-78".
```

Constraints:

- `2 <= number.length <= 100`
- `number` consists of digits and the characters `'-'` and `' '`.
- There are at least **two** digits in `number`.

1455. Check If a Word Occurs As a Prefix of Any Word in a Sentence

Easy

Topics

Companies

Hint

Given a `sentence` that consists of some words separated by a **single space**, and a `searchWord`, check if `searchWord` is a prefix of any word in `sentence`.

Return *the index of the word in* `sentence` *(**1-indexed**) where* `searchWord` *is a prefix of this word*. If `searchWord` is a prefix of more than one word, return the index of the first word **(minimum index)**. If there is no such word return `-1`.

A **prefix** of a string `s` is any leading contiguous substring of `s`.

Example 1:

```
Input: sentence = "i love eating burger", searchWord = "burg"
Output: 4
```

```
Explanation: "burg" is prefix of "burger" which is the 4th wo
rd in the sentence.
```

Example 2:

```
Input: sentence = "this problem is an easy problem", searchWo
rd = "pro"
Output: 2
Explanation: "pro" is prefix of "problem" which is the 2nd an
d the 6th word in the sentence, but we return 2 as it's the m
inimal index.
```

Example 3:

```
Input: sentence = "i am tired", searchWord = "you"
Output: -1
Explanation: "you" is not a prefix of any word in the sentenc
e.
```

Constraints:

- `1 <= sentence.length <= 100`
- `1 <= searchWord.length <= 10`
- `sentence` consists of lowercase English letters and spaces.
- `searchWord` consists of lowercase English letters.

```
class Solution {
public:
    int isPrefixOfWord(string sentence, string searchWord) {
        vector<string> words;
        stringstream ss(sentence);
        string tmp;
        int index = 1;  // Start index from 1 for 1-based index
        while (getline(ss, tmp, ' ')) {
            words.push_back(tmp);
        }

        int j = searchWord.size(); // Use searchWord.size() dire
        for (int i = 0; i < words.size(); i++) {
            if (words[i].substr(0, j) == searchWord) {  // Matc
                return i + 1;  // Return 1-based index
            }
        }

        return -1; // Return -1 if no prefix match is found
    }
};
```

Key Changes

1. **Fixed Prefix Check**: Changed `it.substr(0, j)` to `words[i].substr(0, j)` and compared it with `searchWord` directly.
2. **Adjusted Indexing**: The return value is now `i + 1` to account for 1-based indexing.
3. **Simplified Code**: Removed unnecessary variables and streamlined the logic.

Explanation

- **String Splitting**: Splits the input `sentence` into words using `stringstream` and stores them in the `words` vector.
- **Prefix Check**: Iterates through each word and checks if the `searchWord` is a prefix of the word.
- **Return Value**: Returns the 1-based index of the word that has `searchWord` as a prefix. If none is found, returns `1`.

This revised version should work correctly for all cases and is aligned with typical problem requirements.

1408. String Matching in an Array

Easy

Topics

Companies

Hint

Given an array of string `words`, return *all strings in* `words` *that is a* ***substring*** *of another word*. You can return the answer in **any order**.

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

Example 1:

```
Input: words = ["mass","as","hero","superhero"]
Output: ["as","hero"]
Explanation: "as" is substring of "mass" and "hero" is substr
ing of "superhero".
["hero","as"] is also a valid answer.
```

Example 2:

```
Input: words = ["leetcode","et","code"]
Output: ["et","code"]
Explanation: "et", "code" are substring of "leetcode".
```

Example 3:

```
Input: words = ["blue","green","bu"]
Output: []
```

```
Explanation: No string of words is substring of another strin
g.
```

Constraints:

- `1 <= words.length <= 100`
- `1 <= words[i].length <= 30`
- `words[i]` contains only lowercase English letters.
- All the strings of `words` are **unique**.

```
class Solution {
public:
    vector<string> stringMatching(vector<string>& words) {
        vector<string > v ;

        for(int i=0;i<words.size();i++){
            for(int j=0;j<words.size();j++){
                if(i!=j){
                    if (words[j].find(words[i]) != string::npos
                        v.push_back(words[i]);
                        break;
                    }
                }


            }
        }

        return v ;

    }
};
```

Explanation:

1. **Class and Method Declaration**:
    - `class Solution { ... };`: Defines a class named `Solution`.
    - `vector<string> stringMatching(vector<string>& words) { ... }`: Declares a method `stringMatching` which takes a reference to a vector of strings (`words`) as input and returns a vector of strings (`v`).
2. **Function Implementation**:
    - `vector<string> v;`: Declares an empty vector `v` of strings to store the results.
3. **Nested Loops**:
    - `for(int i = 0; i < words.size(); i++) { ... }`: Outer loop iterates through each word in the `words` vector.
    - `for(int j = 0; j < words.size(); j++) { ... }`: Inner loop also iterates through each word in the `words` vector.
4. **Condition Check**:
    - `if(i != j) { ... }`: Ensures that `i` and `j` are not pointing to the same word in the `words` vector.
5. **Substring Search**:
    - `if(words[j].find(words[i]) != string::npos) { ... }`: Checks if the word at index `j` contains the word at index `i` as a substring.
        - `words[j].find(words[i])` returns the position of the first occurrence of `words[i]` in `words[j]`. If `words[i]` is not found, it returns `string::npos`.
        - If `words[i]` is found within `words[j]`, `string::npos` will not be returned, and the condition will be true.
6. **Result Storage**:
    - `v.push_back(words[i]);`: If `words[i]` is found as a substring in any other word (`words[j] != words[i]` and `words[j].find(words[i]) != string::npos`), `words[i]` is added to the result vector `v`.
    - `break;`: Breaks out of the inner loop after finding the first occurrence of `words[i]` as a substring in any other word, since further checks in the inner

loop are unnecessary once a match is found.

7. **Return**:
    - `return v;`: Returns the vector `v` containing all words from `words` that are substrings of other words in the list.

Summary:

This function `stringMatching` identifies all words in the input vector `words` that are substrings of any other word in the same vector and returns them in a new vector. The nested loops and substring search ensure that each word is compared with every other word exactly once, avoiding unnecessary comparisons after a match is found.

`string::npos` (in simple words is used when substring is not found ..)

`string::npos` is a special constant in C++ that represents "no position". It is used primarily with string operations to indicate that a substring was not found.

What is `string::npos`?

- **`string::npos`**: This is a value that indicates an invalid position or a position that does not exist in a string. The exact value of `string::npos` is the largest possible value for the size of a string (`std::size_t`), but you don't need to remember this technical detail. Instead, just understand that it's a way for functions to say, "I couldn't find what you were looking for."

How is `string::npos` Used?

In C++, when you want to find a substring within a string, you can use the `find` function, which belongs to the `std::string` class. The `find` function returns the index (position) of the first occurrence of the substring if it is found. If the substring is **not found**, `find` returns `string::npos`.

Example to Understand `string::npos`

Let's consider a simple example to illustrate how `string::npos` works:

```
#include <iostream>
#include <string>

int main() {
    std::string text = "Hello, world!";
    std::string substring = "world";
    std::string notFoundSubstring = "planet";

    // Find the position of "world" in "Hello, world!"
    std::size_t position = text.find(substring);

    if (position != std::string::npos) {
        std::cout << "\\"" << substring << "\\" found at posi
tion: " << position << std::endl;
    } else {
        std::cout << "\\"" << substring << "\\" not found." <
< std::endl;
    }

    // Try to find a substring that doesn't exist
    position = text.find(notFoundSubstring);

    if (position != std::string::npos) {
        std::cout << "\\"" << notFoundSubstring << "\\" found
at position: " << position << std::endl;
    } else {
        std::cout << "\\"" << notFoundSubstring << "\\" not f
ound." << std::endl;
    }

    return 0;
}
```

Output:

```
"world" found at position: 7
"planet" not found.
```

Explanation:

1. **Finding an Existing Substring**:
    - `text.find(substring)` looks for `"world"` in the string `"Hello, world!"`.
    - Since `"world"` is present, `find` returns `7` (the index where `"world"` starts in `"Hello, world!"`).
2. **Finding a Non-Existent Substring**:
    - `text.find(notFoundSubstring)` looks for `"planet"` in the string `"Hello, world!"`.
    - `"planet"` is not present in `"Hello, world!"`, so `find` returns `string::npos`.
    - The condition `position != std::string::npos` checks if the position is a valid index. If not, it concludes that the substring was not found.

Why is `string::npos` Important?

`string::npos` is important because it provides a clear and consistent way to check whether a substring exists in a string. Without it, you would need to use more complex checks to see if `find` returned a valid position. Using `string::npos` makes your code easier to read and understand.

`std::size_t` is a data type in C++ that is used to represent the size of any object in bytes. It is an unsigned integer type, which means it can only represent non-negative numbers (zero or positive numbers).

What is `std::size_t`?

- **`std::size_t`**: It is a type defined in the C++ Standard Library. It is designed to hold the size of any object or the result of the `sizeof` operator, which tells you how many bytes a particular data type or object occupies in memory.
- **Unsigned Type**: Since `std::size_t` is unsigned, it can only store non-negative numbers, which makes it ideal for sizes and counts, where negative numbers don't make sense.

Why Use `std::size_t`?

1. **Consistency and Portability**: `std::size_t` ensures that you have a consistent way to represent sizes across different platforms. The actual type of `std::size_t` may vary between platforms (e.g., on some systems it could be an `unsigned int`, on others an `unsigned long`), but using `std::size_t` ensures your code is portable and will compile correctly regardless of the underlying platform.

2. **Memory Management**: `std::size_t` is commonly used when dealing with memory sizes and indexing. For example, it's the return type of the `sizeof` operator, which gives the size of a data type or object in bytes.

Example Uses of `std::size_t`

Let's look at some examples to better understand where `std::size_t` is used:

Example 1: Using `std::size_t` with `sizeof`

```
#include <iostream>

int main() {
    int a = 5;
    std::size_t sizeOfInt = sizeof(a);  // Using sizeof to ge
t the size of an int variable

    std::cout << "Size of int in bytes: " << sizeOfInt << st
d::endl;
    return 0;
}
```

Explanation:

- `sizeof(a)` returns the size of the integer variable `a` in bytes.
- `std::size_t sizeOfInt` stores this value.
- This example demonstrates how `std::size_t` is used to store the result of `sizeof`.

Example 2: Looping with `std::size_t`

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // Using std::size_t for the loop counter
    for (std::size_t i = 0; i < numbers.size(); ++i) {
        std::cout << "Element " << i << ": " << numbers[i] <<
std::endl;
    }

    return 0;
}
```

Explanation:

- `numbers.size()` returns the number of elements in the vector `numbers`. The return type of `numbers.size()` is `std::size_t`.
- Using `std::size_t` for the loop counter `i` ensures the loop variable is compatible with the type returned by `numbers.size()`.
- This avoids any potential compiler warnings or errors about mismatched types when comparing an unsigned and signed integer (such as comparing an `int` with `std::size_t`).

Key Points About `std::size_t`

1. **Unsigned Type**: Since it is unsigned, `std::size_t` can't represent negative numbers. This is perfect for sizes, counts, and indexes, where negative values don't make sense.
2. **Return Type of `sizeof`**: The `sizeof` operator always returns a `std::size_t` because the size of an object can never be negative.

3. **Platform-Dependent Size**: The actual size of `std::size_t` (e.g., whether it's 32 bits or 64 bits) can vary depending on the platform and compiler. It is generally large enough to represent the size of the largest possible object in the address space of the machine.
4. **Safe for Size Comparisons**: When you use `std::size_t` for loop counters or any arithmetic involving sizes, you ensure your code avoids potential pitfalls with signed/unsigned mismatches and is robust across different systems.

In summary, `std::size_t` is a versatile and safe type used for representing sizes, counts, and memory-related values in C++ programming.

1945. Sum of Digits of String After Convert

Easy

Topics

Companies

Hint

You are given a string `s` consisting of lowercase English letters, and an integer `k`.

First, **convert** `s` into an integer by replacing each letter with its position in the alphabet (i.e., replace `'a'` with `1`, `'b'` with `2`, ..., `'z'` with `26`).
Then, **transform** the integer by replacing it with the **sum of its digits**. Repeat the **transform** operation `k` **times** in total.

For example, if `s = "zbax"` and `k = 2`, then the resulting integer would be `8` by the following operations:

- **Convert**: `"zbax" ➝ "(26)(2)(1)(24)" ➝ "262124" ➝ 262124`
- **Transform #1**: `262124 ➝ 2 + 6 + 2 + 1 + 2 + 4 ➝ 17`
- **Transform #2**: `17 ➝ 1 + 7 ➝ 8`

Return *the resulting integer after performing the operations described above.*

Example 1:

```
Input: s = "iiii", k = 1
Output: 36
Explanation: The operations are as follows:
- Convert: "iiii" → "(9)(9)(9)(9)" → "9999" → 9999
- Transform #1: 9999 → 9 + 9 + 9 + 9 → 36
Thus the resulting integer is 36.
```

Example 2:

```
Input: s = "leetcode", k = 2
Output: 6
Explanation: The operations are as follows:
- Convert: "leetcode" → "(12)(5)(5)(20)(3)(15)(4)(5)" → "125
52031545" → 12552031545
- Transform #1: 12552031545 → 1 + 2 + 5 + 5 + 2 + 0 + 3 + 1
+ 5 + 4 + 5 → 33
- Transform #2: 33 → 3 + 3 → 6
Thus the resulting integer is 6.
```

Example 3:

```
Input: s = "zbax", k = 2
Output: 8
```

Constraints:

- `1 <= s.length <= 100`
- `1 <= k <= 10`
- `s` consists of lowercase English letters.

```
class Solution {
public:
    int getLucky(string s, int k) {
        string numericStr = "";

        // Step 1: Convert each character to its corresponding a
        for (char c : s) {
            int position = (c - 'a' + 1); // Convert 'a' to 1,
            numericStr += to_string(position); // Append the nur
        }

        // Step 2: Sum the digits for k transformations
        while (k--) {
            int sum = 0;
            for (char c : numericStr) {
                sum += (c - '0'); // Convert each character bacl
            }
            numericStr = to_string(sum); // Convert the sum bacl
        }

        // Convert the final numericStr to an integer and returr
        return stoi(numericStr);
    }
};
```

Explanation of the Fixed Code

1. **Converting Characters to Numeric Positions**:
    - Loop through each character in the string `s` and convert it to its corresponding numeric position using `(c - 'a' + 1)`. This correctly maps 'a' to 1, 'b' to 2, etc.
    - Append each numeric position to `numericStr` as a string.
2. **Transforming the Numeric Representation**:
    - Use a loop that runs `k` times to sum the digits of the current numeric string.

  - For each character in `numericStr`, convert it to an integer digit by using `(c - '0')` and add it to `sum`.
  - After summing all digits, convert `sum` back to a string for the next iteration.

3. **Final Conversion and Return**:
   - After completing `k` transformations, the numeric result is stored in `numericStr`. Convert this final string back to an integer using `stoi(numericStr)` and return it.

Key Points

- The fixed code ensures that each character is correctly mapped to its alphabetic position.
- It correctly handles string to integer conversions and maintains type consistency throughout the function.
- The logic for transforming the numeric representation for `k` iterations is now correctly implemented.

---

345. Reverse Vowels of a String

Easy

Topics

Companies

Given a string `s`, reverse only all the vowels in the string and return it.

The vowels are `'a'`, `'e'`, `'i'`, `'o'`, and `'u'`, and they can appear in both lower and upper cases, more than once.

Example 1:

```
Input: s = "hello"
Output: "holle"
```

Example 2:

```
Input: s = "leetcode"
Output: "leotcede"
```

Constraints:

- `1 <= s.length <= 3 * 105`
- `s` consist of **printable ASCII** characters.

```
class Solution {
public:
    string reverseVowels(string s) {
        int start = 0;
        int e = s.size() - 1 ;

        while (start<e) {
            if(!isVowel(s[start])){
                start ++;
            }
            else if (!isVowel(s[e])) {
                e--;
            }
            else if (isVowel(s[e]) and isVowel(s[start])) {
                char temp ;
                temp = s[start];
                s[start] = s[e];
                s[e] = temp;
                start++;
                e--;
            }
        }

        return s ;

    }
    bool isVowel(char c){
        if(c =='a' or c == 'e' or c== 'i' or c == 'o' or c=='u'
            return true ;
        }
        else{
            return false;
        }
    }
};
```

APPROACH :

1. create a start pointer and end pointer
2. while(start < end)
    a. check if the s[start] is a vowel
        i. if not then start ++ ;
    b. check if the s[end] is a vowel
        a. if not then end - - ;
    c. if both s[end] == s[start]
        a. swap both s[end] and s[start]

1592. Rearrange Spaces Between Words

Easy

Topics

Companies

Hint

You are given a string `text` of words that are placed among some number of spaces. Each word consists of one or more lowercase English letters and are separated by at least one space. It's guaranteed that `text` **contains at least one word**.

Rearrange the spaces so that there is an **equal** number of spaces between every pair of adjacent words and that number is **maximized**. If you cannot redistribute all the spaces equally, place the **extra spaces at the end**, meaning the returned string should be the same length as `text`.

Return *the string after rearranging the spaces*.

Example 1:

```
Input: text = "  this   is  a sentence "
Output: "this   is   a   sentence"
Explanation: There are a total of 9 spaces and 4 words. We ca
n evenly divide the 9 spaces between the words: 9 / (4-1) = 3
spaces.
```

Example 2:

```
Input: text = " practice   makes   perfect"
Output: "practice   makes   perfect "
Explanation: There are a total of 7 spaces and 3 words. 7 /
(3-1) = 3 spaces plus 1 extra space. We place this extra spac
e at the end of the string.
```

Constraints:

- `1 <= text.length <= 100`
- `text` consists of lowercase English letters and `' '`.
- `text` contains at least one word.

```
#include <string>
#include <vector>
```

```cpp
class Solution {
public:
    string reorderSpaces(string text) {
        int spaceCount = 0;
        vector<string> words;
        string word = "";

        // Count spaces and extract words
        for (char c : text) {
            if (c == ' ') {
                spaceCount++; // Count each space
                if (!word.empty()) {
                    words.push_back(word); // Add the completed
                    word = ""; // Reset for the next word
                }
            } else {
                word += c; // Build the word character by charac
            }
        }
        if (!word.empty()) words.push_back(word); // Add the las

        int numWords = words.size();
        int spacesBetweenWords = numWords > 1 ? spaceCount / (nu
        int extraSpaces = numWords > 1 ? spaceCount % (numWords

        // Build the result string
        string result = "";
        for (int i = 0; i < numWords; ++i) {
            result += words[i]; // Add the word
            if (i < numWords - 1) {
                result += string(spacesBetweenWords, ' '); // Ad
            }
        }
        result += string(extraSpaces, ' '); // Add the remaining
        return result;
    }
};
```

How This Simpler Code Works

1. **Count Spaces and Extract Words**:
    - The code iterates over each character in the input string.
    - It counts the spaces and constructs words.
    - When a space is found, the current word is added to the `words` vector, and the word is reset.
2. **Calculate Spaces Distribution**:
    - `spacesBetweenWords` calculates the number of spaces to be evenly distributed between words.
    - `extraSpaces` calculates how many spaces will be left after even distribution.
3. **Construct the Result**:
    - The result string is built by concatenating words with the calculated spaces in between.
    - Any leftover spaces (`extraSpaces`) are added at the end of the result string.

Benefits of This Approach

- **Simpler Logic**: The code is straightforward, focusing on counting, calculating, and constructing the result step-by-step.
- **No `stringstream` Needed**: This approach avoids using `stringstream` for extracting words, making it easier to understand.
- **Edge Cases Handled**: It correctly handles cases where there is only one word or no words at all.

459. Repeated Substring Pattern

Easy

Topics

Companies

Given a string `s`, check if it can be constructed by taking a substring of it and appending multiple copies of the substring together.

Example 1:

```
Input: s = "abab"
Output: true
Explanation: It is the substring "ab" twice.
```

Example 2:

```
Input: s = "aba"
Output: false
```

Example 3:

```
Input: s = "abcabcabcabc"
Output: true
Explanation: It is the substring "abc" four times or the subs
tring "abcabc" twice.
```

Constraints:

- `1 <= s.length <= 104`
- `s` consists of lowercase English letters.

```
class Solution {
public:
    bool repeatedSubstringPattern(string s) {
        int n = s.size();

        // Loop through each possible substring length from 1 to
        for (int len = 1; len <= n / 2; ++len) {
            if (n % len == 0) {  // Check if the substring lengt
                string substring = s.substr(0, len);  // Get the
                string repeated = "";  // Create a new string by

                // Repeat the substring enough times to match th
                for (int i = 0; i < n / len; ++i) {
                    repeated += substring;
                }

                // Check if the constructed string matches the o
                if (repeated == s) {
                    return true;
                }
            }
        }

        return false;  // Return false if no valid repeating pat
    }
};
```

Explanation of the Corrected Code

1. **Iterate Over Possible Substring Lengths**:
   - Loop through each possible length of the substring from `1` to `n/2` (where `n` is the length of the string `s`).
2. **Check Divisibility**:
   - If the string length `n` is divisible by `len`, it means `len` could be the length of the repeating substring.
3. **Construct and Compare**:

    - Extract a substring of length `len` and repeatedly append it to form a new string.
    - Compare this new string to the original string `s`.
4. **Return Result**:
    - If a repeating pattern is found that matches the original string `s`, return `true`.
    - If no such pattern is found after all iterations, return `false`.

Key Changes Made

- The code now correctly initializes variables.
- Logical conditions have been corrected to properly check substring patterns.
- An efficient loop to test all possible repeating substrings is implemented.
- The function now correctly returns `true` or `false` based on whether the string can be constructed by repeating a substring.

This revised code will correctly determine if a string can be constructed by repeating a substring and is more efficient and readable

---

482. License Key Formatting

Easy

Topics

Companies

You are given a license key represented as a string `s` that consists of only alphanumeric characters and dashes. The string is separated into `n + 1` groups by `n` dashes. You are also given an integer `k`.

We want to reformat the string `s` such that each group contains exactly `k` characters, except for the first group, which could be shorter than `k` but still must contain at least one character. Furthermore, there must be a dash inserted between two groups, and you should convert all lowercase letters to uppercase.

Return *the reformatted license key*.

Example 1:

```
Input: s = "5F3Z-2e-9-w", k = 4
Output: "5F3Z-2E9W"
Explanation: The string s has been split into two parts, each
part has 4 characters.
Note that the two extra dashes are not needed and can be remo
ved.
```

Example 2:

```
Input: s = "2-5g-3-J", k = 2
Output: "2-5G-3J"
Explanation: The string s has been split into three parts, ea
ch part has 2 characters except the first part as it could be
shorter as mentioned above.
```

Constraints:

- `1 <= s.length <= 105`
- `s` consists of English letters, digits, and dashes `'-'`.
- `1 <= k <= 104`

```
class Solution {
public:
    string licenseKeyFormatting(string s, int k) {
        string alpha = "";

        // 1. Remove dashes and convert all letters to uppercase
        for (char c : s) {
            if (c != '-') {  // Skip dashes, add all other chara
                alpha += toupper(c);
            }
        }

        int n = alpha.size();
        string res = "";

        // 2. Calculate the length of the first group.
        int firstGroupLength = n % k;
        int index = 0;

        // 3. Add the first group if it has a non-zero length.
        if (firstGroupLength > 0) {
            res += alpha.substr(index, firstGroupLength);
            index += firstGroupLength;
            if (index < n) {
                res += "-";  // Add a dash after the first grou
            }
        }

        // 4. Add the remaining groups of size 'k'.
        while (index < n) {
            res += alpha.substr(index, k);
            index += k;
            if (index < n) {
                res += "-";  // Add a dash after every group of
            }
        }

        return res;
    }
};
```

Explanation of the Corrected Code

1. **Filter and Convert to Uppercase**:

  - Loop through each character in the string `s`. Skip dashes (`'-'`), and convert all other characters to uppercase using `toupper`.
2. **Compute the Size of the First Group**:
  - The first group may be shorter than `k` if the total length is not a multiple of `k`. We use `firstGroupLength = n % k` to determine its size.
3. **Build the Formatted String**:
  - Start by adding the first group to `res` if `firstGroupLength` is greater than `0`. Add a dash after it unless it's the only group.
  - Continue adding groups of size `k` to `res` and appending a dash after each until all characters are formatted.
  - We only add a dash when `index` is still less than `n` to avoid trailing dashes.
4. **Return the Formatted String**:
  - The formatted string `res` is returned, which represents the desired license key formatting.

Benefits of This Approach

- **Clear and Simple**: This code avoids modifying strings in place unnecessarily and clearly handles each section of the formatting logic.
- **Efficient**: Only passes through the input string once to build the filtered `alpha` string, and once more to format the result, making it O(n) in time complexity.
- **Correct Handling of Edge Cases**: Ensures that no trailing dashes are added and that the first group can correctly be shorter than `k`.

optimised code :

```
class Solution {
public:
    string licenseKeyFormatting(string s, int k) {
        string result;
        int count = 0;
        // Traverse the string in reverse order
        for (int i = s.size() - 1; i >= 0; --i) {
            if (s[i] != '-') {  // Skip dashes
                if (count == k) {  // Insert a dash after every
                    result += '-';
                    count = 0;  // Reset count after adding a da
                }
                result += toupper(s[i]);  // Append uppercase ch
                count++;
            }
        }

        reverse(result.begin(), result.end());  // Reverse the r

        return result;
    }
};
```

Explanation of the Optimized Code

1. **Iterate in Reverse**:
   - Start from the end of the string and move to the beginning. This way, we don't need to worry about how many characters are in the first group.
2. **Add Characters and Dashes**:
   - For every character that isn't a dash (`'-'`), convert it to uppercase and add it to the `result` string.
   - After every `k` characters added to `result`, insert a dash (`'-'`), then reset the count to `0`.
3. **Reverse the Result**:

- Once the loop is complete, reverse the `result` string to transform it from its reverse form to the correct order.

4. **Return the Result**:
    - The formatted string is now ready and correctly formatted.

Benefits of This Approach

- **Single Pass**: Only a single pass through the string is needed, plus a final reversal of the result string.
- **No Conditional Checks for the First Group**: By building the string from the end, we don't need to worry about how the first group differs from others.
- **Improved Clarity and Simplicity**: The logic is straightforward, reducing the potential for bugs related to index management and conditional checks.

Time Complexity

- The optimized solution still has a time complexity of O(n), where n is the length of the input string `s`.

  O(n)O(n)

  nn

- The space complexity is O(n) as well, due to storing the result in the new string.

  O(n)

Overall, the optimized approach is simpler, easier to understand, and avoids unnecessary operations while maintaining the same time complexity.