Check Palindrome

```
class Solution {
    isPalindrome(s){
        let left = 0, right = s.length - 1;

        while(left < right) {
            if(s[left++] != s[right--])
                return 0;
        }

        return 1;
    }
}
```

Check one string is rotation of other

```
function check_rotation(s, goal){
    if (s.length != goal.length)
        return false;

    let q1 = []
    for(let i=0;i<s.length;i++)
        q1.push(s[i])

    let q2 = []
    for(let i=0;i<goal.length;i++)
        q2.push(goal[i])

    let k = goal.length
    while (k--){
        let ch = q2[0]
        q2.shift()
        q2.push(ch)
        if (JSON.stringify(q2) == JSON.stringify(q1))
            return true
    }

    return false
}
```

```
function areRotations( str1, str2) {
    return (str1.length == str2.length) &&
        ((str1 + str1).indexOf(str2) != -1);
}
```

```
function areRotations( str1, str2) {
    if (str1.length !== str2.length) {
        return false;
    } else {
    for (let i = 0; i < str1.length; i++) {
        if (str1[i] === str2[0]) {
        //checking suffix of s1 with pref of s2
            if (str1.substr(i) === str2.substr(0, str1.length - i)) {
                // checking prefix of s1 with suffix of s2
                if (str1.substr(0, i) === str2.substr(str1.length - i)) {
                    return true;
                }
            }
        }
    }
}
return false;
}
```

Check one string is shuffled of other


```
function isShuffledSubstring(A, B) {
    var n = A.length, m = B.length;
    if (n > m)
        return false;
    else {
        // Sort string A
        A = A.split('').sort().join('');
        // Traverse string B
        for (var i = 0; i < m; i++) {
            // Return false if (i+n-1 >= m) doesn't satisfy
            if (i + n - 1 >= m)
                return false;
            var str = [];
            for (var j = 0; j < n; j++)
                str.push(B[i + j]);
            str = str.sort()
            if (str.join('') == A)
                return true;
        }
    }
}
```



```
function compare(arr1,arr2) {
    for(let i = 0; i < 256; i++)
        if (arr1[i] != arr2[i])
            return false;
    return true;
}

function search(pat,txt) {
    let M = pat.length, N = txt.length;

    let countP = new Array(256);
    let countTW = new Array(256);
    for(let i = 0; i < 256; i++) {
        countP[i] = 0;
        countTW[i] = 0;
    }
    for(let i = 0; i < 256; i++) {
        countP[i] = 0;
        countTW[i] = 0;
    }
    for(let i = 0; i < M; i++) {
        (countP[pat[i].charCodeAt(0)])++;
        (countTW[txt[i].charCodeAt(0)])++;
    }
    for(let i = M; i < N; i++) {
        if (compare(countP, countTW))
            return true;
        (countTW[txt[i].charCodeAt(0)])++;
        countTW[txt[i - M].charCodeAt(0)]--;
    }
    if (compare(countP, countTW))
        return true;
    return false;
}
```

Facebook 5 | Google 3 | Amazon 2 | Apple 2 | Adobe 2 | Walmart Global Tech 2

38. Count and Say

Medium | 3343 | 7018 | Add to List

The **count-and-say** sequence is a sequence of digit strings defined by the recursive formula:

- `countAndSay(1) = "1"`
- `countAndSay(n)` is the way you would "say" the digit string from `countAndSay(n-1)`, which is then converted into a different digit string.

To determine how you "say" a digit string, split it into the **minimal** number of substrings such that each substring contains exactly **one** unique digit. Then for each substring, say the number of digits, then say the digit. Finally, concatenate every said digit.

For example, the saying and conversion for digit string `"3322251"`:

"3322251"
two 3's, three 2's, one 5, and one 1
2 3 + 3 2 + 1 5 + 1 1
"23321511"

Given a positive integer `n`, return *the* `n^th` *term of the* ***count-and-say*** *sequence.*

Example 1:

```
Input: n = 1
Output: "1"
Explanation: This is the base case.
```

Example 2:

```
Input: n = 4
Output: "1211"
Explanation:
countAndSay(1) = "1"
countAndSay(2) = say "1" = one 1 = "11"
countAndSay(3) = say "11" = two 1's = "21"
countAndSay(4) = say "21" = one 2 + one 1 = "12" + "11" = "1211"
```

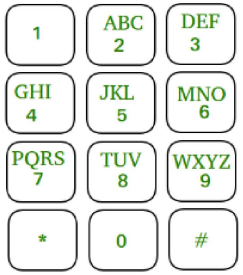
Constraints:

- `1 <= n <= 30`

Accepted 807.1K | Submissions 1.5M

```cpp
class Solution {
public:
    string countAndSay(int n) {
        if(n == 1) return "1";
        if(n == 2) return "11";

        string sn = "11";

        for(int i=3; i<=n; i++) {
            sn += "@";      // so that last char gets added
            int count = 1;
            string temp = "";

            for(int j=1; j<sn.size(); j++) {
                if(sn[j-1] == sn[j])
                    count++;
                else{
                    temp += to_string(count);
                    temp += sn[j-1];
                    count = 1;
                }
            }
            sn = temp;
        }
        return sn;
    }
};
```

Given a sentence in the form of a string, convert it into its equivalent mobile numeric keypad sequence.



Examples :

Input: *GEEKSFORGEEKS*
Output: *4333355777733366677743333557777*
Explanation: *For obtaining a number, we need to press a number corresponding to that character for a number of times equal to the position of the character. For example, for character E, press number 3 two times and accordingly.*

Input : *HELLO WORLD*
Output : *44335555556660966677755 53*

```
function printSequence(arr,input) {
    let output = "";
    let n = input.length;
    for (let i = 0; i < n; i++) {
        if (input[i] == ' ')
            output = output + "0".charCodeAt(0);
        else {
            // Calculating index for each character
            let position = input[i].charCodeAt(0) - 'A'.charCodeAt(0);
            output = output + arr[position];
        }
    }
    return output;
}
// Driver Function
let str = ["2", "22", "222",    "3", "33", "333",
"4", "44", "444","5", "55", "555",
"6", "66", "666","7", "77", "777", "7777",
"8", "88", "888","9", "99", "999", "9999" ]

let input = "GEEKSFORGEEKS";
document.write(printSequence(str, input));
```


Roman Number to Integer

Easy Accuracy: 43.31% Submissions: 110K+ Points: 2

Given a string in roman no format (s) your task is to convert it to an integer . Various symbols and their values are given below.
I 1
V 5
X 10
L 50
C 100
D 500
M 1000

Example 1:

Input:
s = V
Output: 5

Example 2:

Input:
s = III
Output: 3

Approach: A number in Roman Numerals is a string of these symbols written in descending order(e.g. M's first, followed by D's, etc.). However, in a few specific cases, to avoid four characters being repeated in succession(such as IIII or XXXX), **subtractive notation** is often used as follows:

- **I** placed before **V** or **X** indicates one less, so four is **IV** (one less than 5) and 9 is IX (one less than 10).
- **X** placed before **L** or **C** indicates ten less, so forty is **XL** (10 less than 50) and 90 is **XC** (ten less than a hundred).
- **C** placed before **D** or **M** indicates a hundred less, so four hundred is **CD** (a hundred less than five hundred) and nine hundred is **CM** (a hundred less than a thousand).

Algorithm to convert Roman Numerals to Integer Number:

1. *Split the Roman Numeral string into Roman Symbols (character).*
2. *Convert each symbol of Roman Numerals into the value it represents.*
3. *Take symbol one by one from starting from index 0:*
    1. *If current value of symbol is greater than or equal to the value of next symbol, then add this value to the running total.*
    2. *else subtract this value by adding the value of next symbol to the running total.*

```cpp
class Solution {
    public:
    int romanToDecimal(string &str) {
        unordered_map<char, int> mp;
        mp.insert({'I', 1});
        mp.insert({'V' ,5});
        mp.insert({'X' ,10});
        mp.insert({'L', 50});
        mp.insert({'C' ,100});
        mp.insert({ 'D' , 500});
        mp.insert({ 'M' , 1000});

        int decimal = 0;

        for(int i=0; i<str.size(); i++){
            if(mp[str[i]] < mp[str[i+1]])
                decimal -= mp[str[i]];
            else
                decimal += mp[str[i]];
        }

        return decimal;
    }
};
```

```javascript
var roman = new Map() ;
roman.set('I', 1);
roman.set('V', 5);
roman.set('X', 10);
roman.set('L', 50);
roman.set('C', 100);
roman.set('D', 500);
roman.set('M', 1000);

function romanToInt( s) {
    var sum = 0;
    var n = s.length;
    for (i = 0; i < n; i++) {
        if (i != n - 1 && roman.get(s.charAt(i)) < roman.get(s.charAt(i + 1))) {
            sum += roman.get(s.charAt(i + 1)) - roman.get(s.charAt(i));  // IX = 40..
            i++;
        } else {
            sum += roman.get(s.charAt(i));
        }
    }
    return sum;
}
```

Facebook 26 | Apple 14 | Google 12

Adobe 11 | Amazon 9 | Microsoft 5

Bloomberg 5 | Uber 4 | Quora 3 | SAP 3

14. Longest Common Prefix

Easy 14889 4088 Add to List Share

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string `""`.

Example 1:

```
Input: strs = ["flower","flow","flight"]
Output: "fl"
```

Example 2:

```
Input: strs = ["dog","racecar","car"]
Output: ""
Explanation: There is no common prefix among the input strings.
```

```javascript
/**
 * @param {string[]} strs
 * @return {string}
 */
var longestCommonPrefix = function(strs) {
    if (!strs.length) return "";

    let prefix = strs[0];
    let best_len = Number.MAX_SAFE_INTEGER;

    for (let i = 1; i < strs.length; i++) {
        let j = 0;
        while (j < strs[i].length && j < prefix.length && prefix[j] === strs[i][j]) {
            j++;
        }

        if (j < best_len) {
            prefix = strs[i].substr(0, j);
            best_len = j;
        }
    }

    return prefix;
};
```

165. Compare Version Numbers

Medium 1993 2482 Add to List

Given two version numbers, `version1` and `version2`, compare them.

Version numbers consist of **one or more revisions** joined by a dot `'.'`. Each revision consists of **digits** and may contain leading **zeros**. Every revision contains **at least one character**. Revisions are **0-indexed from left to right**, with the leftmost revision being revision 0, the next revision being revision 1, and so on. For example `2.5.33` and `0.1` are valid version numbers.

To compare version numbers, compare their revisions in **left-to-right order**. Revisions are compared using their **integer value ignoring any leading zeros**. This means that revisions `1` and `001` are considered **equal**. If a version number does not specify a revision at an index, then **treat the revision as** `0`. For example, version `1.0` is less than version `1.1` because their revision 0s are the same, but their revision 1s are `0` and `1` respectively, and `0 < 1`.

Return the following:

- If `version1 < version2`, return `-1`.
- If `version1 > version2`, return `1`.
- Otherwise, return `0`.

Example 1:

```
Input: version1 = "1.01", version2 = "1.001"
Output: 0
Explanation: Ignoring leading zeroes, both "01" and "001" represent the same integer "1".
```

Example 2:

```
Input: version1 = "1.0", version2 = "1.0.0"
Output: 0
Explanation: version1 does not specify revision 2, which means it is treated as "0".
```

Example 3:

```
Input: version1 = "0.1", version2 = "1.1"
Output: -1
Explanation: version1's revision 0 is "0", while version2's revision 0 is "1". 0 < 1, so version1 < version2.
```

```javascript
/**
 * @param {string} version1
 * @param {string} version2
 * @return {number}
 */
var compareVersion = function(version1, version2) {
    let i = 0, j = 0;
    const n1 = version1.length, n2 = version2.length;
    let num1 = 0, num2 = 0;

    while (i < n1 || j < n2) {
        // Getting digitNum for version1
        while (i < n1 && version1[i] !== '.') {
            num1 = num1 * 10 + (version1[i] - '0');
            i++;
        }
        // Getting digitNum for version2
        while (j < n2 && version2[j] !== '.') {
            num2 = num2 * 10 + (version2[j] - '0');
            j++;
        }

        if (num1 > num2)
            return 1;
        if (num1 < num2)
            return -1;

        i++, j++;                // Move both pointers past the dots
        num1 = 0, num2 = 0;      // Reset digitNum
    }

    return 0;
};
```

```
var compareVersion = function(version1, version2) {
    const arr1 = version1.split('.');
    const arr2 = version2.split('.');
    const n = Math.max(arr1.length, arr2.length);

    for (let i = 0; i < n; i++) {
        const num1 = parseInt(arr1[i]) || 0;
        const num2 = parseInt(arr2[i]) || 0;

        if (num1 > num2)
            return 1;
        if (num1 < num2)
            return -1;
    }

    return 0;
};
```


Facebook 10 | Amazon 6 | Snapchat 3

Google 2 | Adobe 2 | Uber 2

67. Add Binary

Easy | 8361 | 829 | Add to List | Share

Given two binary strings `a` and `b`, return *their sum as a binary string*.

Example 1:

```
Input: a = "11", b = "1"
Output: "100"
```

Example 2:

```
Input: a = "1010", b = "1011"
Output: "10101"
```


Constraints:

- `1 <= a.length, b.length <= 10^4`
- `a` and `b` consist only of `'0'` or `'1'` characters.
- Each string does not contain leading zeros except for the zero itself.

Accepted 1,196,973 | Submissions 2,278,892

```cpp
class Solution {
public:
    string addBinary(string a, string b) {
        string s = "";

        int c = 0, i = a.size() - 1, j = b.size() - 1;
        while(i >= 0 || j >= 0 || c == 1) {
            c += i >= 0 ? a[i --] - '0' : 0;
            c += j >= 0 ? b[j --] - '0' : 0;
            s = char(c % 2 + '0') + s;  // for rev
            c /= 2;
        }

        return s;
    }
};
```



```javascript
/**
 * @param {string} a
 * @param {string} b
 * @return {string}
 */
var addBinary = function(a, b) {
    let s = "";
    let carry = 0;
    let i = a.length - 1;
    let j = b.length - 1;

    while (i >= 0 || j >= 0 || carry > 0) {
        const digitA = i >= 0 ? parseInt(a[i--]) : 0;
        const digitB = j >= 0 ? parseInt(b[j--]) : 0;
        const sum = digitA + digitB + carry;

        s = (sum % 2) + s;    // fro reverse
        carry = Math.floor(sum / 2);
    }

    return s;
};
```


Min Number of Flips

Easy Accuracy: 48.58% Submissions: 41K+ Points: 2

Given a binary string, that is it contains only 0s and 1s. We need to make this string a sequence of alternate characters by flipping some of the bits, our goal is to minimize the number of bits to be flipped.

Example 1:

Input:
S = "001"
Output: 1
Explanation:
We can flip the 0th bit to 1 to have 101.

Example 2:

Input:
S = "0001010111"
Output: 2
Explanation: We can flip the 1st and 8th bit
bit to have "0101010101"
101.

```cpp
// "0001010111" it can be either
// "0101010101" or "1010101010" now match wrt it
int minFlips (string s) {
    int n = s.size();
    int zeroFirst = 0, oneFirst = 0;
    for(int i=0; i<n; i++) {
        if(i%2==0 and s[i]=='1') zeroFirst++;
        if(i%2==1 and s[i]=='0') zeroFirst++;

        if(i%2==0 and s[i]=='0') oneFirst++;
        if(i%2==1 and s[i]=='1') oneFirst++;
    }

    return min(oneFirst, zeroFirst);
}
```


Bloomberg 33 | Google 19 | Microsoft 14 | Amazon 12

Facebook 11 | Cisco 8 | Oracle 6 | Snapchat 5

Apple 5 | tiktok 5 | Adobe 3 | Uber 3 | Intuit 2

Cruise Automation 2 | ByteDance 2 | Walmart Global Tech 2

Sumologic 2

394. Decode String

Medium 11380 516 Add to List Share

Given an encoded string, return its decoded string.

The encoding rule is: `k[encoded_string]`, where the `encoded_string` inside the square brackets is being repeated exactly `k` times. Note that `k` is guaranteed to be a positive integer.

You may assume that the input string is always valid; there are no extra white spaces, square brackets are well-formed, etc. Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, `k`. For example, there will not be input like `3a` or `2[4]`.

The test cases are generated so that the length of the output will never exceed 10^5.

Example 1:

```
Input: s = "3[a]2[bc]"
Output: "aaabcbc"
```

Example 2:

```
Input: s = "3[a2[c]]"
Output: "accaccacc"
```

Example 3:

```
Input: s = "2[abc]3[cd]ef"
Output: "abcabccdcdcdef"
```

```
var decodeString = function(str) {
/*
we have 4 possibilities -
1) opening braces -> new sequence starts, so add curr string and curr number to the stack,
    reassign both to initial values
2) closing braces -> the sequence is over, it is time to create a substring
by getting prev string and prev number from the stack, add prev string(repeated prev num times)
to curr string
3) if it is number add to curr num
4) if it is char add to curr string
*/
    let stack = [];
    let currStr = '';
    let currNum = 0;

    for (let i = 0; i < str.length; i ++) {
        if (str[i] === '[') {
            stack.push(currStr);    // this would be prev on ']'
            stack.push(currNum);
            currStr = '';
            currNum = 0;
        }
        else if (str[i] === ']') {
            let prevNum = stack.pop();
            let prevStr = stack.pop();
            currStr = prevStr + currStr.repeat(prevNum);    // "" + a*3  then aaa + 2*bc
        }
        else if (str[i] >= '0' && str[i] <= '9') {
            currNum = currNum * 10 + Number(str[i]);
        }
        else {
            currStr += str[i];
        }
    }

    return currStr;
}
```


Second most repeated string in a sequence

Easy Accuracy: 45.95% Submissions: 60K+ Points: 2

Given a sequence of strings, the task is to find out the second most repeated (or frequent) string in the given sequence.

Note: No two strings are the second most repeated, there will be always a single string.

Example 1:

Input:
N = 6
arr[] = {aaa, bbb, ccc, bbb, aaa, aaa}
Output: bbb
Explanation: "bbb" is the second most occurring string with frequency 2.

Example 2:

Input:
N = 6
arr[] = {geek, for, geek, for, geek, aaa}
Output: for
Explanation: "for" is the second most occurring string with frequency 2.

Your Task:
You don't need to read input or print anything. Your task is to complete the function **secFrequent()** which takes the string array arr[] and its size N as inputs and returns the second most frequent string in the array.

Expected Time Complexity: O(N*max(|S_i|)).
Expected Auxiliary Space: O(N*max(|S_i|)).

```cpp
class Solution {
public:
    string secFrequent(string arr[], int n) {
        unordered_map<string, int> mp;
        for (int i = 0; i < n; i++) {
            mp[arr[i]]++;
        }

        int first = -1, second = 0;
        string firstStr = "", secondStr = "";
        for (auto it : mp) {
            if (it.second > first) {
                // order matters pahle sec=first and then update first
                second = first;
                first = it.second;
                secondStr = firstStr;
                firstStr = it.first;
            }
            else if (it.second > second) {
                second = it.second;
                secondStr = it.first;
            }
        }

        return secondStr;
    }
};
```



```javascript
class Solution {
    secFrequent(arr, n) {
        const mp = {};
        for (let i = 0; i < n; i++) {
            if (!mp[arr[i]]) {
                mp[arr[i]] = 1;
            } else {
                mp[arr[i]]++;
            }
        }

        let first = -1, second = 0;
        let firstStr = "", secondStr = "";

        for (const key in mp) {
            if (mp.hasOwnProperty(key)) {
                const value = mp[key];
                if (value > first) {
                    second = first;
                    first = value;
                    secondStr = firstStr;
                    firstStr = key;
                } else if (value > second) {
                    second = value;
                    secondStr = key;
                }
            }
        }

        return secondStr;
    }
}
```

string encode

Interview problems

122 Views 0 Replies

```
string encode(string &message)
{
   //  Write your code here.
   string temp;
   int count = 1;
   char word;
   for(int i = 0;i<message.length(); i++){
       word = message[i];
       if(word == message[i+1]){
          count++;
       }
       else{
           temp += word;
           temp += to_string(count);
           count = 1;
       }
   }
   return temp;

}
```


String Subsequence Game

Medium Accuracy: 38.87% Submissions: 6K+ Points: 4

Given a string return all unique possible subsequences which start with vowel and end with consonant. A String is a subsequence of a given String, that is generated by deleting some character of a given string without changing its order.
NOTE: Return all the unique subsequences in lexicographically sorted order.

Example 1:

Input: S = "abc"
Output: "ab", "ac", "abc"
Explanation: "ab", "ac", "abc" are the all possible subsequences which start with vowel and end with consonant.

Example 2:

Input: S = "aab"
Output: "ab", "aab"
Explanation: "ab", "aab" are the all possible subsequences which start with vowel and end with consonant.

Your Task:
You dont need to read input or print anything. Complete the function **allPossileSubsequences()** which takes S as input parameter and returns all possible subsequences which start with vowel and end with consonant.

Expected Time Complexity: $O(n*\log n*2^n)$
Expected Auxiliary Space: $O(2^n)$

Constraints:
$1 <= |S| <= 18$

```cpp
class Solution {
public:
    bool isVowel(char c) {
        return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
    }

    void subseq(string& s, string temp, int idx, vector<string>& st) {
        if (idx >= s.size()) {
            st.push_back(temp);
            return;
        }
        subseq(s, temp + s[idx], idx + 1, st);
        subseq(s, temp, idx + 1, st);
    }

    set<string> allPossibleSubsequences(string s) {
        vector<string> st;
        subseq(s, "", 0, st);
        set<string> res;

        for (auto subs : st) {
            if (isVowel(subs[0]) && !isVowel(subs.back())) {
                res.insert(subs);
            }
        }

        return res;
    }
};
```



```javascript
class Solution {
    isVowel(c) {
        return c === 'a' || c === 'e' || c === 'i' || c === 'o' || c === 'u';
    }

    subseq(s, temp, idx, st) {
        if (idx >= s.length) {
            st.push(temp);
            return;
        }
        this.subseq(s, temp + s[idx], idx + 1, st);
        this.subseq(s, temp, idx + 1, st);
    }

    allPossibleSubsequences(s) {
        const st = [];
        this.subseq(s, '', 0, st);
        const res = new Set();

        for (const subs of st) {
            if (subs.length > 0 && this.isVowel(subs[0]) && !this.isVowel(subs[subs.length - 1])) {
                res.add(subs);
            }
        }

        return res;
    }
}
```

Permutations of a given string

Medium Accuracy: 34.65% Submissions: 231K+ Points: 4

Given a string **S**. The task is to print all **unique** permutations of the given string in lexicographically sorted order.

Example 1:

Input: ABC
Output:
ABC ACB BAC BCA CAB CBA
Explanation:
Given string ABC has permutations in 6
forms as ABC, ACB, BAC, BCA, CAB and CBA .

Example 2:

Input: ABSG
Output:
ABGS ABSG AGBS AGSB ASBG ASGB BAGS
BASG BGAS BGSA BSAG BSGA GABS GASB
GBAS GBSA GSAB GSBA SABG SAGB SBAG
SBGA SGAB SGBA
Explanation:
Given string ABSG has 24 permutations.

Your Task:
You don't need to read input or print anything. Your task is to complete the function **find_permutaion()** which takes the string S as input parameter and returns a vector of string in lexicographical order.

Expected Time Complexity: O(n! * n)
Expected Space Complexity: O(n! * n)


```cpp
class Solution {
    public:
    int n;
    void solve(int idx, string &s, set<string> &st) {
        if(idx >= n) {
            st.insert(s);
            return;
        }

        for(int i=idx; i<n; i++) {
            swap(s[i], s[idx]);
            solve(idx+1, s, st);
            // backtracking
            swap(s[i], s[idx]);
        }
    }
    vector<string>find_permutation(string s) {
        n = s.size();
        set<string> st;
        solve(0, s, st);
        vector<string> ans(st.begin(), st.end());

        return ans;
    }
};
```

```javascript
/**  In JavaScript, strings are immutable, which means you can't
     directly swap characters using swap(s[i], s[idx]).
     You need to convert the string to an array, perform the swap,
     and then join the array back into a string
The Array.from(result) statement is used to convert the set back to an array for the output

class Solution {
    find_permutation(S) {
        const result = new Set();

        function swap(str, i, j) {
            const strArray = str.split('');
            [strArray[i], strArray[j]] = [strArray[j], strArray[i]];
            return strArray.join('');
        }

        function permute(str, start) {
            if (start === str.length - 1) {
                result.add(str);
                return;
            }

            for (let i = start; i < str.length; i++) {
                const swappedStr = swap(str, start, i);
                permute(swappedStr, start + 1);
            }
        }

        permute(S, 0);
        // Sort the permutations array
        const sortedPermutations = Array.from(result).sort();

        return sortedPermutations;
    }
}
```


Recursively remove all adjacent duplicates

Given a string, recursively remove adjacent duplicate characters from the string. The output string should not have any adjacent duplicates. See the following examples.

Examples:

Input: *azxxzy*
Output: *ay*

- *First "azxxzy" is reduced to "azzy".*
- *The string "azzy" contains duplicates,*
- *so it is further reduced to "ay".*

Input: *geeksforgeeg*
Output: *gksfor*

- *First "geeksforgeeg" is reduced to "gksforgg".*
- *The string "gksforgg" contains duplicates,*
- *so it is further reduced to "gksfor".*

Input: *caaabbbaacdddd*
Output: *Empty String*

Input: *acaaabbbacdddd*
Output: *acac*

```cpp
string removeDuplicates(string s) {
    string ans = "";
    int i = 0, n = s.size();
    while (i < n) {
        // If the current character is the same as the next one, skip duplicates
        if (i < n - 1 && s[i] == s[i + 1]) {
            while (i < n - 1 && s[i] == s[i + 1])
                i++;
        } else {
            ans.push_back(s[i]);
        }
        i++;
    }
    return ans;
}
// Recursive function to remove duplicates until no more can be removed
string remove(string s) {
    string newS = removeDuplicates(s);
    if (newS.size() == s.size())
        return s;
    else
        return remove(newS);
}
```



```javascript
class Solution {
    rremove(s) {
        let modified = false;
        let result = '';

        for (let i = 0; i < s.length; ) {
            let j = i + 1;
            while (j < s.length && s[i] === s[j]) {
                modified = true;
                j++;
            }

            if (!modified) {
                result += s[i];
            }

            modified = false;
            i = j;
        }

        if (result === s) {
            return result; // No more duplicates to remove
        } else {
            return this.rremove(result); // Recurse with the new string
        }
    }
}
```

Recursively print all sentences that can be formed from list of word lists

Given a list of word lists How to print all sentences possible taking one word from a list at a time via recursion?

Example:

```
Input: {{"you", "we"},
        {"have", "are"},
        {"sleep", "eat", "drink"}}

Output:
  you have sleep
  you have eat
  you have drink
  you are sleep
  you are eat
  you are drink
  we have sleep
  we have eat
  we have drink
  we are sleep
  we are eat
  we are drink
```

We strongly recommend minimizing your browser and try this yourself first.

The idea is based on a simple depth-first traversal. We start from every word of the first list as the first word of an output sentence, then recur for the remaining lists.

Below is the C++ implementation of the above idea. In the below implementation, the input list of the list is considered as a 2D array. If we take a closer look, we can observe that the code is very close to the DFS of graph.


```cpp
void solve(string arr[3][3], string str, int row, int R, int C) {
    if (row == R) {
        cout << str << endl;
        return;
    }
    // add currRow word and Recur for next row
    for (int i = 0; i < C; i++) {
        if (!arr[row][i].empty())
            solve(arr, str + " " + arr[row][i], row + 1, R, C);
    }
}

int main() {
    int R = 3, C = 3;
    string arr[3][3] = {{"you", "we"},
                        {"have", "are"},
                        {"sleep", "eat", "drink"}};

    solve(arr, "", 0, R, C);

    return 0;
}
```

Split the binary string into substrings with equal number of 0s and 1s

Prateek_Aggarwal

Given a binary string **str** of length **N**, the task is to find the maximum count of consecutive substrings **str** can be divided into such that all the substrings are balanced i.e. they have equal number of **0s** and **1s**. If it is not possible to split **str** satisfying the conditions then print **-1**.

Example:

Input: *str = "0100110101"*
Output: *4*
The required substrings are "01", "0011", "01" and "01".
Input: *str = "0111100010"*
Output: *3*

Input: *str = "0000000000"*
Output: *-1*

Approach: Initialize **count = 0** and traverse the string character by character and keep track of the number of **0s** and **1s** so far, whenever the count of **0s** and **1s** become equal increment the count. As in the given question, if it is not possible to split string then on that time count of 0s must not be equal to count of 1s then return **-1** else print the value of count after the traversal of the complete string. Below is the implementation of the above approach:

```
int maxSubStr(string str, int n) {
    int ans = 0;
    stack<char> st;

    for (int i = 0; i < n; i++) {
        if (st.empty() || st.top() == str[i])
            st.push(str[i]);
        else {
            st.pop();
            if (st.empty())
                ans++;
        }
    }

// If the st isnt empty, it means string cant be split into substrs
    if (!st.empty())
        return -1;

    return ans;
}
```



```
int maxSubStr(string str, int n) {
    int count0 = 0, count1 = 0;

    int cnt = 0;
    for (int i = 0; i < n; i++) {
        if (str[i] == '0')  count0++;
        else  count1++;
        if (count0 == count1) {
            cnt++;
        }
    }
    if (count0 != count1)
        return -1;

    return cnt;
}
```

Next Permutation

Medium Accuracy: 40.66% Submissions: 113K+ Points: 4

Implement the next permutation, which rearranges the list of numbers into Lexicographically next greater permutation of list of numbers. If such arrangement is not possible, it must be rearranged to the lowest possible order *i.e.* sorted in an ascending order. You are given an list of numbers **arr[ ]** of size **N**.

Example 1:

Input: N = 6
arr = {1, 2, 3, 6, 5, 4}
Output: {1, 2, 4, 3, 5, 6}
Explaination: The next permutation of the given array is {1, 2, 4, 3, 5, 6}.

Example 2:

Input: N = 3
arr = {3, 2, 1}
Output: {1, 2, 3}
Explaination: As arr[] is the last permutation. So, the next permutation is the lowest one.

Your Task:
You do not need to read input or print anything. Your task is to complete the function **nextPermutation()** which takes N and arr[ ] as input parameters and returns a list of numbers containing the next permutation.

Expected Time Complexity: O(N)
Expected Auxiliary Space: O(1)

Constraints:
$1 \leq N \leq 10000$


```cpp
class Solution{
public:
    vector<int> nextPermutation(int n, vector<int> arr){

        int downfall = -1;  // getting idx of 3
        for(int i=n-2; i>=0; i--) {
            if(arr[i] < arr[i+1]) {
                downfall = i;
                break;
            }
        }
        if(downfall == -1){
            reverse(arr.begin(), arr.end());
            return arr;
        }

        int justGreater = -1; // getting idx of 4
        for(int i=n-1; i>=0; i--) {
            if(arr[i] > arr[downfall]) {
                justGreater = i;
                break;
            }
        }
        // swapped (3, 4)
        swap(arr[downfall], arr[justGreater]);

        // reverse from 6 ie downfall + 1
        reverse(arr.begin() + downfall + 1, arr.end());

        return arr;
    }
};
```

```
class Solution {
    nextPermutation(arr, n) {
        let downfall = -1;
        for (let i = n - 2; i >= 0; i--) {
            if (arr[i] < arr[i + 1]) {
                downfall = i;
                break;
            }
        }
        if (downfall === -1) {
            arr.reverse();
            return arr;
        }

        let justGreater = -1;
        for (let i = n - 1; i >= 0; i--) {
            if (arr[i] > arr[downfall]) {
                justGreater = i;
                break;
            }
        }

        [arr[downfall], arr[justGreater]] = [arr[justGreater], arr[downfall]];
        arr.splice(downfall + 1, n - downfall - 1, ...arr.slice(downfall + 1).reverse());

        return arr;
    }
}
```


1328. Break a Palindrome

Medium 2154 720 Add to List Share

Given a palindromic string of lowercase English letters `palindrome`, replace **exactly one** character with any lowercase English letter so that the resulting string is **not** a palindrome and that it is the **lexicographically smallest** one possible.

Return *the resulting string. If there is no way to replace a character to make it not a palindrome, return an* ***empty string****.*

A string `a` is lexicographically smaller than a string `b` (of the same length) if in the first position where `a` and `b` differ, `a` has a character strictly smaller than the corresponding character in `b`. For example, `"abcc"` is lexicographically smaller than `"abcd"` because the first position they differ is at the fourth character, and `'c'` is smaller than `'d'`.

Example 1:

```
Input: palindrome = "abccba"
Output: "aaccba"
Explanation: There are many ways to make "abccba" not a palindrome,
such as "zbccba", "aaccba", and "abacba".
Of all the ways, "aaccba" is the lexicographically smallest.
```

Example 2:

```
Input: palindrome = "a"
Output: ""
Explanation: There is no way to replace a single character to make "a"
not a palindrome, so return an empty string.
```

```cpp
class Solution {
public:
    string breakPalindrome(string palindrome) {
        int  n = palindrome.size();
        if(n < 2) return "";

        for(int i = 0; i < n/2; i++) {
            if(palindrome[i] != 'a') {
                palindrome[i] = 'a';
                return palindrome;
            }
        }
        // case all same
        palindrome[n-1] = 'b';

        return palindrome;
    }
};
```



```javascript
var breakPalindrome = function(palindrome) {
    const n = palindrome.length;
    if (n < 2) return "";

    const chars = palindrome.split('');

    for (let i = 0; i < Math.floor(n / 2); i++) {
        if (chars[i] !== 'a') {
            chars[i] = 'a';
            return chars.join('');
        }
    }

    // Case when all characters are 'a'
    chars[n - 1] = 'b';

    return chars.join('');
};
```


Parenthesis Checker

Easy Accuracy: 28.56% Submissions: 472K+ Points: 2

Given an expression string **x**. Examine whether the pairs and the orders of {,},(,),[,] are correct in exp. For example, the function should return 'true' for exp = [()]{}{[()()]()} and 'false' for exp = [(]).

Note: The drive code prints "balanced" if function return true, otherwise it prints "not balanced".

Example 1:

Input:
{{[]}}
Output:
true
Explanation:
{ ([ ]) }. Same colored brackets can form balanced pairs, with 0 number of unbalanced bracket.

Example 2:

Input:
()
Output:
true
Explanation:
(). Same bracket can form balanced pairs, and here only 1 type of bracket is present and in balanced way.

Example 3:

Input:
([]
Output:
false
Explanation:
([]. Here square bracket is balanced but the small bracket is not balanced and Hence , the output will be unbalanced.

```cpp
class Solution{
    public:
    //Function to check if brackets are balanced or not.
    bool ispar(string str){
        stack<char>stk;
        for(int i=0;i<str.length();i++){
            if(str[i]=='{' || str[i]=='[' || str[i]=='('){
                stk.push(str[i]);
            }
            else{
                if(!stk.empty()){
                    char top = stk.top();
                    if( top=='{' && str[i]=='}' ||
                        top=='[' && str[i]==']' ||
                        top=='(' && str[i]==')' )
                        stk.pop();
                    else return false;
                }
                else  return false;
            }
        }
        return stk.empty();
    }
};
```

```javascript
class Solution {
    //Function to check if brackets are balanced or not.
    ispar(x) {
        const stk = [];
        for (let i = 0; i < x.length; i++) {
            if (x[i] === '{' || x[i] === '[' || x[i] === '(') {
                stk.push(x[i]);
            } else {
                if (stk.length > 0) {
                    const top = stk[stk.length - 1];
                    if ((top === '{' && x[i] === '}') ||
                        (top === '[' && x[i] === ']') ||
                        (top === '(' && x[i] === ')')) {
                        stk.pop();
                    } else {
                        return false;
                    }
                } else {
                    return false;
                }
            }
        }
        return stk.length === 0;
    }
}
```


Facebook 16 | Amazon 15 | Microsoft 13 | Apple 9

Adobe 4 | Google 4 | Bloomberg 3 | Uber 3 | C3 IoT 3

Walmart Global Tech 3 | Oracle 3 | ByteDance 2 | Tesla 2

22. Generate Parentheses

Medium 18940 762 Add to List Share

Given `n` pairs of parentheses, write a function to *generate all combinations of well-formed parentheses.*

Example 1:

```
Input: n = 3
Output: ["((()))","(()())","(())()","()(())","()()()"]
```

Example 2:

```
Input: n = 1
Output: ["()"]
```

Constraints:

- `1 <= n <= 8`

```cpp
class Solution {
public:
    void solve(int open, int close, int limit, string par, vector<string> &ans) {
        if(open == limit and close == limit) {
            ans.push_back(par);
            return;
        }

        if(open < limit) {
            solve(open + 1, close, limit, par + '(', ans);
        }
        if(close < open) {
            solve(open, close + 1, limit, par + ')', ans);
        }
    }

    vector<string> generateParenthesis(int n) {
        string par = "";
        vector<string> ans;
        solve(0, 0, n, par, ans);
        return ans;
    }
};
```



```javascript
var generateParenthesis = function(n) {
    const ans = [];

    function solve(open, close, limit, par) {
        if (open === limit && close === limit) {
            ans.push(par);
            return;
        }

        if (open < limit) {
            solve(open + 1, close, limit, par + '(');
        }
        if (close < open) {
            solve(open, close + 1, limit, par + ')');
        }
    }

    solve(0, 0, n, '');
    return ans;
};
```

Count the Reversals

Medium Accuracy: 51.88% Submissions: 54K+ Points: 4

Given a string **S** consisting of only opening and closing curly brackets **'{'** and **'}'**, find out the minimum number of reversals required to convert the string into a balanced expression.

A reversal means changing **'{'** to **'}'** or vice-versa.

Example 1:

Input:
S = "}{{}}{{{"
Output: 3
Explanation: One way to balance is: "**{{{}}{}}**". There is no balanced sequence that can be formed in lesser reversals.

Example 2:

Input:
S = "{{}{{{}{{}}{{"
Output: -1
Explanation: There's no way we can balance this sequence of braces.

```javascript
function countRev(s) {
    const size = s.length;
    if (size & 1)
        return -1;

    let open = 0, close = 0;
    for (let i = 0; i < size; i++) {
        const c = s[i];
        if (c === '{') {
            open++;
        } else if (open > 0 && c === '}') {
            open--;
        } else {
            close++;
        }
    }

    const openRev = Math.ceil(open / 2);
    const closeRev = Math.ceil(close / 2);

    return openRev + closeRev;
}

// Example usage
const inputString = "{{}{{}{{}}{{";
console.log(countRev(inputString)); // Outputs: 2
```

```cpp
int countRev (string s) {

    int size = s.size();
    if(size&1)
        return -1;

    int open = 0, close = 0;
    for(char c: s) {
        if(c == '{')
            open++;
        else if(open > 0 and c == '}')
            open--;
        else
            close++;


    }
    int openRev = ceil(open/2.0);
    int closeRev = ceil(close/2.0);

    return openRev + closeRev;
}
```

2730. Find the Longest Semi-Repetitive Substring

Medium 218 65 Add to List Share

You are given a **0-indexed** string `s` that consists of digits from `0` to `9`.

A string `t` is called a **semi-repetitive** if there is at most one consecutive pair of the same digits inside `t`. For example, `0010`, `002020`, `0123`, `2002`, and `54944` are semi-repetitive while `00101022`, and `1101234883` are not.

Return *the length of the longest semi-repetitive substring inside* `s`.

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

Example 1:

```
Input: s = "52233"
Output: 4
Explanation: The longest semi-repetitive substring is "5223", which
starts at i = 0 and ends at j = 3.
```

Example 2:

```
Input: s = "5494"
Output: 4
Explanation: s is a semi-reptitive string, so the answer is 4.
```

Example 3:

```
Input: s = "1111111"
Output: 2
Explanation: The longest semi-repetitive substring is "11", which
starts at i = 0 and ends at j = 1.
```

```cpp
class Solution {
    public:
    int longestSemiRepetitiveSubstring(string s) {
        int n = s.size(), ans = 0;

        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                string t = s.substr(i, j - i + 1);
                int cnt = 0;

                for (int k = 1; k < t.size(); k++) {
                    if (t[k] == t[k - 1])
                        cnt++;
                    }
                    if (cnt <= 1)
                        ans = max(ans, j - i + 1);
                }
            }

        return ans;
    }
};
```

```cpp
class Solution {
public:
    int longestSemiRepetitiveSubstring(string s) {
       int i = 0 , j = 0;

        int ans = 0; int len = 1;
        if(s.size() <= 2)
            return s.size();

        for( j=1; j<s.size(); j++) {
            len++;
            if(s[j] == s[j-1]){
                len = j-i+1;
                i = j;
            }
            if(len > ans)
                ans = len;
        }
        return ans;
    }
};
```

```cpp
class Solution {
public:
    // upsolve optimized approach
    int longestSemiRepetitiveSubstring(string s) {
        int n = s.size();
        int start = 0, end = 1, count = 0, ans = 1;

        while(end < n) {
            if(s[end] == s[end-1]) {
                count++;
                if(count == 2) {
                    // removing first dup from window
                    while(count != 1) {
                        if(s[start] == s[start+1])
                            count--;
                        start++;
                    }
                }
            }
            ans = max(ans, end-start+1);
            end++;
        }

        return ans;
    }
};
```

```javascript
var longestSemiRepetitiveSubstring = function(s) {
    let n = s.length;
    let start = 0, end = 1, count = 0, ans = 1;

    while (end < n) {
        if (s[end] === s[end - 1]) {
            count++;
            if (count === 2) {
                // Removing first duplicate from window
                while (count !== 1) {
                    if (s[start] === s[start + 1])
                        count--;
                    start++;
                }
            }
        }
        ans = Math.max(ans, end - start + 1);
        end++;
    }

    return ans;
};
```

Minimum Swaps for Bracket Balancing

Easy Accuracy: 37.36% Submissions: 32K+ Points: 2

Share your Interview, Campus or Work Experience to win GFG Swag Kits and much more!

You are given a string S of 2N characters consisting of N '[' brackets and N ']' brackets. A string is considered balanced if it can be represented in the form S2[S1] where S1 and S2 are balanced strings. We can make an unbalanced string balanced by swapping **adjacent** characters.
Calculate the minimum number of swaps necessary to make a string balanced.

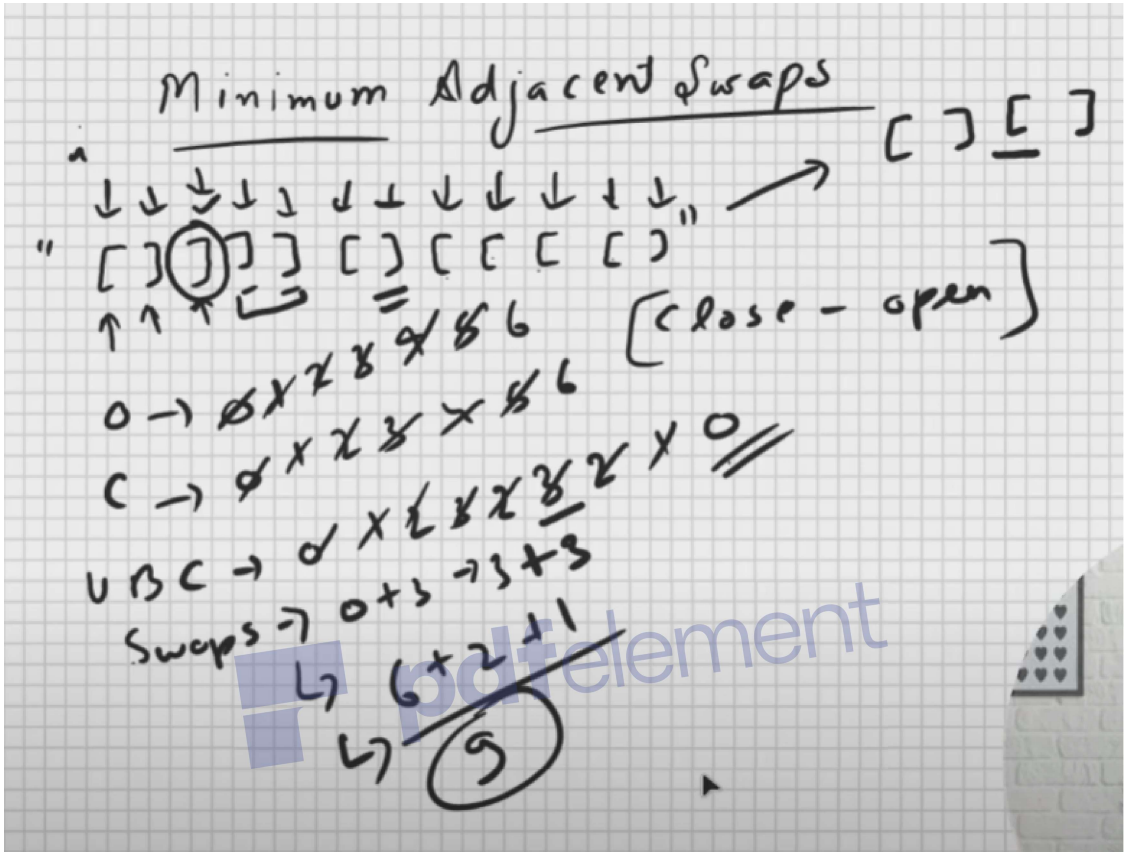
Note - Strings S1 and S2 can be empty.

Example 1:

Input : []][][
Output : 2
Explanation :
First swap: Position 3 and 4
[][][
Second swap: Position 5 and 6
[][][]

Example 2:

Input : [[][]]
Output : 0
Explanation:
String is already balanced.



```cpp
class Solution{
public:
    int minimumNumberOfSwaps(string s){
        int open = 0, close = 0;
        int unbalanced = 0;
        int noOfSwaps = 0;

        for(char c: s) {
            if(c == '[') {
                open++;
                if(unbalanced > 0) {
                    noOfSwaps += unbalanced;
                    unbalanced--;   // 1 fixed
                }
            }
            else if(c == ']') {
                close++;
                unbalanced = (close - open);
            }
        }

        return noOfSwaps;
    }
};
```



```javascript
function countRev(s) {
    const size = s.length;
    if (size & 1) {
        return -1;
    }

    let open = 0, close = 0;
    for (let i = 0; i < size; i++) {
        const c = s[i];
        if (c === '{') {
            open++;
        } else if (open > 0 && c === '}') {
            open--;
        } else {
            close++;
        }
    }

    const openRev = Math.ceil(open / 2);
    const closeRev = Math.ceil(close / 2);

    return openRev + closeRev;
}

// Example usage
const inputString = "{{}{{}{{}}{{";
console.log(countRev(inputString)); // Outputs: 2
```

```cpp
const int prime = 31;
const int mod = 1e9 + 9;
vector<int> rabinKarp(string text, string pattern) {
    int n = text.size(), m = pattern.size();
    vector<int> result;

    int power = 1;
    for (int i = 0; i < m - 1; i++)
        power = (power * prime) % mod;

    int patternHash = 0, textHash = 0;

    // Calculate hash value for the pattern and the first window in the text
    for (int i = 0; i < m; i++) {
        patternHash = (patternHash * prime + pattern[i]) % mod;
        textHash = (textHash * prime + text[i]) % mod;
    }

    // Slide the pattern over the text and check for matches
    for (int i = 0; i <= n - m; i++) {
        if (patternHash == textHash) {
            // Verify character by character if there is a match
            bool match = true;
            for (int j = 0; j < m; j++) {
                if (text[i + j] != pattern[j]) {
                    match = false;
                    break;
                }
            }
            if (match) {
                result.push_back(i);
            }
        }

        // Update the rolling hash value for the next window
        if (i < n - m) {
            textHash = (textHash - text[i] * power) % mod;
            if (textHash < 0) {
                textHash += mod;
            }
            textHash = (textHash * prime + text[i + m]) % mod;
        }
    }

    return result;
}
```

```javascript
class Solution {
    search(pat, txt) {
        const val = (ch) => ch.charCodeAt(0) - 'a'.charCodeAt(0) + 1;
        const p = 31, mod = 1e9 + 7;
        const m = pat.length, n = txt.length;

        const p_pow = new Array(Math.max(m, n)).fill(1);
        for (let i = 1; i < p_pow.length; i++)
            p_pow[i] = (p_pow[i - 1] * p) % mod;

        const hash = new Array(n + 1).fill(0);
        for (let i = 0; i < n; i++)
            hash[i + 1] = (hash[i] + val(txt[i]) * p_pow[i]) % mod;

        let pHash = 0;
        for (let i = 0; i < m; i++)
            pHash = (pHash + val(pat[i]) * p_pow[i]) % mod;

        const ans = [];
        for (let i = 0; i < n - m + 1; i++) {
            let tHash = (hash[i + m] - hash[i] + mod) % mod;
            if (tHash === pHash * p_pow[i] % mod) {
                if (txt.substring(i, i + m) === pat) {
                    ans.push(i + 1);
                }
            }
        }
        if (ans.length === 0)
            ans.push(-1);
        return ans;
    }
}
```

1316. Distinct Echo Substrings

Hard 293 195 Add to List Share

Return the number of **distinct** non-empty substrings of `text` that can be written as the concatenation of some string with itself (i.e. it can be written as `a + a` where `a` is some string).

Example 1:

```
Input: text = "abcabcabc"
Output: 3
Explanation: The 3 substrings are "abcabc", "bcabca" and
"cabcab".
```

Example 2:

```
Input: text = "leetcodeleetcode"
Output: 2
Explanation: The 2 substrings are "ee" and "leetcodeleetcode".
```

Constraints:

- `1 <= text.length <= 2000`
- `text` has only lowercase English letters.


```cpp
class Solution {
public:
    int distinctEchoSubstrings(string text) {
        int n = text.size();
        unordered_set<string> st;

        // trying every len possible strs
        for(int len=1; len<=n/2; len++) {
            int count =  0;

            for(int left=0, right=len; right<n; left++, right++) {
                if(text[left] == text[right]) {
                    count++;
                }
                else {
                    count = 0;
                }

                if(count==len) {
                    string possible = text.substr(left + 1, len);
                    if (st.find(possible) == st.end()) {
                        st.insert(possible);
                    }
                    // think it as we skipped that char
                    count--;
                }
            }
        }

        return st.size();
    }
};
```

```cpp
class Solution {
    long long *hash;
    long long *pow;
    int radix = 26;
    long long mod = 1e9 + 7;
public:
    void precompute(string text, int n) {
        hash = new long long[n], pow = new long long[n];

        pow[0] = 1;
        for (int i = 1; i < n; i++) {
            hash[i] = ((hash[i - 1] % mod) * radix + (text[i])) % mod;
            pow[i] = (pow[i - 1] * radix) % mod;
        }
    }
    long long getHash(int l, int r) {
        long long h1 = (hash[r] % mod - (hash[l] % mod * pow[r - l] % mod) % mod + mod) % mod;
        return h1;
    }
    int distinctEchoSubstrings(string text) {
        int n = text.size();
        unordered_set<long long> st;
        precompute(text, n);
        // trying every len possible strs
        for (int len = 1; len <= n / 2; len++) {
            int count = 0;
            for (int left = 0, right = len; right < n; left++, right++) {
                if (text[left] == text[right])
                    count++;
                else count = 0;

                if (count == len) {
                    long long hValue = getHash(left, right);
                    st.insert(hValue);
                    count--;
                }
            }
        }
        delete[] hash;
        delete[] pow;

        return st.size();
    }
};
```


1147. Longest Chunked Palindrome Decomposition

Hard 604 28 Add to List Share

You are given a string `text`. You should split it to k substrings `(subtext_1, subtext_2, ..., subtext_k)` such that:

- `subtext_i` is a **non-empty** string.
- The concatenation of all the substrings is equal to `text` (i.e., `subtext_1 + subtext_2 + ... + subtext_k == text`).
- `subtext_i == subtext_k - i + 1` for all valid values of `i` (i.e., `1 <= i <= k`).

Return the largest possible value of `k`.

Example 1:

```
Input: text = "ghiabcdefhelloadamhelloabcdefghi"
Output: 7
Explanation: We can split the string on "(ghi)(abcdef)(hello)(adam)(hello)(abcdef)(ghi)".
```

Example 2:

```
Input: text = "merchant"
Output: 1
Explanation: We can split the string on "(merchant)".
```

Example 3:

```
Input: text = "antaprezatepzapreanta"
Output: 11
Explanation: We can split the string on "(a)(nt)(a)(pre)(za)(tep)(za)(pre)(a)(nt)(a)".
```

```cpp
class Solution {
    const int Mod = int(1e9) + 123;
    const int Base = 26;
public:
    int helper(const string& s, int l, int r) {
        if (l >= r)  return 0;
        long long prefix_hash = 0;
        long long suffix_hash = 0;
        long long pow = 1;

        for (int i = 0; i < (r - l) / 2; ++i) {

            prefix_hash = (prefix_hash + (s[l + i] - 'a') * pow) % Mod;
            suffix_hash = (suffix_hash * Base + (s[r - i - 1] - 'a')) % Mod;
            pow = (pow * Base) % Mod;
            // explicit manual check O(N)
            if (prefix_hash == suffix_hash) {
                if (s.substr(l, i + 1) == s.substr(r - i - 1, i + 1)) {
                    int newleft = l + i + 1, newRight = r - i - 1;
                    return 2 + helper(s, newleft, newRight);
                    // "ghi abcdefabcdef ghi" now call for "abcdefabcdef"
                }
            }
        }
        // no equal chunks
        return 1;
    }

    int longestDecomposition(string text) {
        return helper(text, 0, text.size());
    }
};
// TC: O(N^2) and SC: O(N) stackSpace
```


Longest Prefix Suffix

Medium Accuracy: 27.91% Submissions: 93K+ Points: 4

Given a string of characters, find the length of the longest proper prefix which is also a proper suffix.

NOTE: Prefix and suffix can be overlapping but they should not be equal to the entire string.

Example 1:

Input: s = "abab"
Output: 2
Explanation: "ab" is the longest proper prefix and suffix.

Example 2:

Input: s = "aaaa"
Output: 3
Explanation: "aaa" is the longest proper prefix and suffix.

Your task:
You do not need to read any input or print anything. The task is to complete the function **lps()**, which takes a string as input and returns an integer.

Expected Time Complexity: O(|s|)
Expected Auxiliary Space: O(|s|)

```cpp
// a b c d e a b c d => 4 (abcde)
// a b c d e a b a d => 4 (ab)

class Solution {
public:
    int lps(string s) {
        int n = s.size();
        int *lpsArray = new int[n];
        int lenj = 0;
        int i = 1;

        lpsArray[0] = 0;
        while (i < n) {
            if (s[lenj] == s[i]) {
                lpsArray[i] = lenj + 1;
                lenj++, i++;
            } else {
                if (lenj != 0) {
                    lenj = lpsArray[lenj - 1];
                } else {
                    lpsArray[i] = 0;
                    i++;
                }
            }
        }

        int maxLPS = lpsArray[n-1]; // we have to take whole str
        delete[] lpsArray;

        return maxLPS;
    }
};
```



```javascript
class Solution {
    lps(s) {
        const n = s.length;
        const lpsArray = new Array(n).fill(0);
        let lenj = 0;
        let i = 1;

        lpsArray[0] = 0;
        while (i < n) {
            if (s[lenj] === s[i]) {
                lpsArray[i] = lenj + 1;
                lenj++, i++;
            } else {
                if (lenj !== 0) {
                    lenj = lpsArray[lenj - 1];
                } else {
                    lpsArray[i] = 0;
                    i++;
                }
            }
        }

        const maxLPS = lpsArray[n - 1];

        return maxLPS;
    }
}
```

Search Pattern (KMP-Algorithm)

Medium Accuracy: 45.04% Submissions: 14K+ Points: 4

Given two strings, one is a text string, **txt** and other is a pattern string, **pat**. The task is to print the indexes of all the occurences of pattern string in the text string. For printing, Starting Index of a string should be taken as 1.

Example 1:

Input:
txt = "batmanandrobinarebat", pat = "bat"
Output: 1 18
Explanation: The string "bat" occurs twice in txt, one starts are index 1 and the other at index 18.

Example 2:

Input:
txt = "abesdu", pat = "edu"
Output: -1
Explanation: There's not substring "edu" present in txt.

Your Task:
You don't need to read input or print anything. Your task is to complete the function **search()** which takes the string txt and the string pat as inputs and returns an array denoting the start indices (1-based) of substring pat in the string txt.
Note: Return an empty list incase of no occurences of pattern. Driver will print -1 in this case.

Expected Time Complexity: O(|txt|).
Expected Auxiliary Space: O(|txt|).


```
class Solution {
    findlps(str, lps) {
        for (let l = 0, r = 1; r < str.length; ) {
            if (str[l] === str[r]) {
                lps[r++] = ++l;
            } else {
                if (l > 0) l = lps[l - 1];
                else r++;
            }
        }
    }

    search(pat, txt) {
        const lps = new Array(pat.length).fill(0);
        const answer = [];
        this.findlps(pat, lps);

        for (let i_txt = 0, i_pat = 0; i_txt < txt.length; ) {
            if (txt[i_txt] === pat[i_pat]) {
                i_txt++;
                i_pat++;
                if (i_pat === pat.length) {
                    answer.push(i_txt - pat.length + 1);
                    i_pat = 0;
                    if (i_txt - pat.length >= 0)
                        i_txt -= pat.length - 1;
                }
            } else {
                if (i_pat > 0)
                    i_pat = lps[i_pat - 1];
                else
                    i_txt++;
            }
        }

        return answer;
    }
}
```

```cpp
class Solution {
public:
    vector<int> search(string pat, string txt) {
        int n = pat.size(), m = txt.size();
        vector<int> lps(n, 0), ans;
        int lenj = 0, i = 1;
        while (i < n) {
            if (pat[lenj] == pat[i]) {
                lenj++;
                pat[i] = lenj;
                i++;
            } else {
                if (i != 0)
                    i = lps[lenj - 1];
                else {
                    lps[i] = 0;
                    i++;
                }
            }
        }
        i = 0, j = 0;
        while (i < m) {
            if (txt[i] != pat[j]) {
                if (j > 0)
                    j = lps[j - 1];
                else
                    i++;

            } else {
                i++,  j++;
                if (j == n) {
                    ans.push_back(i - j + 1);
                    j = lps[j - 1];
                }
            }
        }

        return ans;
    }
};
```

Count occurrences of a given word in a 2-d array

Hard Accuracy: 63.65% Submissions: 14K+ Points: 8

Find the number of occurrences of a given search word in a 2d-Array of characters where the word can go up, down, left, right, and around 90-degree bends.

Note: While making a word you can use one cell only once.

Example 1:

Input:
R = 4, C = 5
mat = {{S,N,B,S,N},
{B,A,K,E,A},
{B,K,B,B,K},
{S,E,B,S,E}}
target = SNAKES
Output:
3
Explanation:
S N B S N
B A K E A
B K B B K
S E B S E
Total occurrence of the word is 3
and denoted by color.

Example 2:

Input:
R = 3, C = 3
mat = {{c,a,t},
{a,t,c},
{c,t,a}}
target = cat
Output:
5
Explanation: Same explanation
as first example.

```cpp
class Solution{
    public:
    int m, n, strSize;
    int dfs(int i, int j, int idx, vector<vector<char>> &mat, string target) {
        if(i<0 || j<0 || i>=m || j>=n || mat[i][j] == '.' || target[idx] != mat[i][j])
            return 0;
        if(idx == strSize - 1) return 1;

        char save = mat[i][j];
        mat[i][j] = '.';   // marked visited

        int down  = dfs(i+1, j, idx+1, mat, target);
        int right = dfs(i, j+1, idx+1, mat, target);
        int up    = dfs(i-1, j, idx+1, mat, target);
        int left  = dfs(i, j-1, idx+1, mat, target);

        mat[i][j] = save;  // bactrack

        return down + right + up + left;
    }

    int findOccurrence(vector<vector<char>> &mat, string target) {
        m = mat.size(), n = mat[0].size(), strSize = target.size();

        int count = 0;
        for(int i=0; i<m; i++) {
            for(int j=0; j<n; j++) {
                if(mat[i][j] == target[0]) {
                    count += dfs(i, j, 0, mat, target);
                }
            }
        }

        return count;
    }
};
```

```javascript
class Solution {
    dfs(i, j, idx, mat, target) {
        if (i < 0 || j < 0 || i >= this.m || j >= this.n ||
                mat[i][j] === '.' || target[idx] !== mat[i][j])
            return 0;

        if (idx === this.strSize - 1) return 1;

        const save = mat[i][j];
        mat[i][j] = '.';   // visited

        const down = this.dfs(i + 1, j, idx + 1, mat, target);
        const right = this.dfs(i, j + 1, idx + 1, mat, target);
        const up = this.dfs(i - 1, j, idx + 1, mat, target);
        const left = this.dfs(i, j - 1, idx + 1, mat, target);

        mat[i][j] = save;  // backtrack

        return down + right + up + left;
    }

    findOccurrence(mat, target) {
        this.m = mat.length;
        this.n = mat[0].length;
        this.strSize = target.length;

        let count = 0;
        for (let i = 0; i < this.m; i++) {
            for (let j = 0; j < this.n; j++) {
                if (mat[i][j] === target[0]) {
                    count += this.dfs(i, j, 0, mat, target);
                }
            }
        }
        return count;
    }
}
```

Find the string in grid

Medium Accuracy: 22.88% Submissions: 36K+ Points: 4

Given a 2D grid of **n*m** of characters and a **word**, find all occurrences of given word in grid. A word can be matched in **all 8 directions** at any point. Word is said to be found in a direction if all characters match in this direction (not in zig-zag form). The 8 directions are, **horizontally left**, **horizontally right**, **vertically up**, **vertically down**, and **4 diagonal directions**.

Note: The returning list should be **lexicographically smallest**. If the word can be found in multiple directions starting from the same coordinates, the list should contain the coordinates only once.

Example 1:

Input:
grid = {{a,b,c},{d,r,f},{g,h,i}},
word = "abc"
Output:
{{0,0}}
Explanation:
From (0,0) we can find "abc" in horizontally right direction.

Example 2:

Input:
grid = {{a,b,a,b},{a,b,e,b},{e,b,e,b}}
word = "abe"
Output:
{{0,0},{0,2},{1,0}}
Explanation:
From (0,0) we can find "abe" in right-down diagonal.
From (0,2) we can find "abe" in left-down diagonal.
From (1,0) we can find "abe" in horizontally right direction.


```cpp
class Solution {
    int m, n, wordLen;
    vector<vector<int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1},
                                      {-1, -1}, {-1, 1}, {1, -1}, {1, 1}};
public:
    bool isValid(int x, int y) {
        return x >= 0 && y >= 0 && x < m && y < n;
    }

    bool searchWordFrom(int x, int y, vector<vector<char>>& grid, string word) {
        if (grid[x][y] != word[0]) return false;

        for (int d = 0; d < 8; d++) {
            int newX = x, newY = y;
            int i;
            for (i = 1; i < wordLen; i++) {
                newX += directions[d][0];
                newY += directions[d][1];
                if (!isValid(newX, newY) || grid[newX][newY] != word[i])
                    break;
            }
            if (i == wordLen) return true;
        }
        return false;
    }

    vector<vector<int>> searchWord(vector<vector<char>>& grid, string word) {
        m = grid.size(), n = grid[0].size(), wordLen = word.length();
        vector<vector<int>> result;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (searchWordFrom(i, j, grid, word)) {
                    result.push_back({i, j});
                }
            }
        }

        return result;
    }
};
```

aybe incorrect

```
class Solution {
    isValid(x, y, m, n) {
        return x >= 0 && y >= 0 && x < m && y < n;
    }

    dfs(grid, x, y, s, idx, dir) {
        const m = grid.length, n = grid[0].length;
        if (idx === s.length) return true;

        if (!this.isValid(x, y, m, n) || grid[x][y] !== s[idx])
            return false;

        const dx = [-1, -1, -1, 0, 0, 1, 1, 1];
        const dy = [-1, 0, 1, -1, 1, -1, 0, 1];

        if (this.dfs(grid, x + dx[dir], y + dy[dir], s, idx + 1, dir))
            return true;

        return false;
    }

    searchWord(grid, s) {
        const ans = [];
        const n = grid.length, m = grid[0].length;
        for (let i = 0; i < n; ++i) {
            for (let j = 0; j < m; ++j) {
                if (grid[i][j] === s[0]) {
                    for (let dir = 0; dir < 8; ++dir) {
                        if (this.dfs(grid, i, j, s, 0, dir)) {
                            ans.push([i, j]);
                            break;
                        }
                    }
                }
            }
        }

        return ans;
    }
}
```


Longest Palindrome in a String

Medium Accuracy: 23.2% Submissions: 266K+ Points: 4

Given a string S, find the longest palindromic substring in S. **Substring of string S:** S[i . . . . j] where $0 \leq i \leq j < \text{len}(S)$. **Palindrome string:** A string that reads the same backward. More formally, S is a palindrome if reverse(S) = S. **In case of conflict**, return the substring which occurs first (with the least starting index).

Example 1:

Input:
S = "aaaabbaa"
Output: aabbaa
Explanation: The longest Palindromic substring is "aabbaa".

Example 2:

Input:
S = "abc"
Output: a
Explanation: "a", "b" and "c" are the longest palindromes with same length. The result is the one with the least starting index.

Your Task:
You don't need to read input or print anything. Your task is to complete the function **longestPalin()** which takes the string S as input and returns the longest palindromic substring of S.

Expected Time Complexity: $O(|S|^2)$.
Expected Auxiliary Space: $O(1)$.

```cpp
class Solution {
    public:
    string longestPalin(string s) {
        int n = s.length();
        int maxLength = 1;
        int start = 0;

        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                // Check if the substr from i to j is a palindrome
                bool isPalindrome = true;
                for (int l = i, r = j; l < r; l++, r--) {
                    if (s[l] != s[r]) {
                        isPalindrome = false;
                        break;
                    }
                }

                if (isPalindrome && j - i + 1 > maxLength) {
                    maxLength = j - i + 1;
                    start = i;
                }
            }
        }

        return s.substr(start, maxLength);
    }
};
```



```cpp
class Solution {
    public:
    string longestPalin (string S) {
        int start = 0, n = S.length(), len = 1;
        for(int i=0; i<n; i++) {
            // odd length expandable
            int left = i - 1, right = i + 1;
            while(left >= 0 and right < n and S[left] == S[right]){
                if(len < right - left + 1){
                    len = right - left + 1;
                    start = left;
                }
                left--, right++;
            }

            // even length expandable
            left = i, right = i+1;
            while(left >= 0 and right < n and S[left] == S[right]){
                if(len < right - left + 1){
                    len = right - left + 1;
                    start = left;
                }
                left--, right++;
            }
        }

        return S.substr(start, len);
    }
};
```

```javascript
class Solution {
    longestPalin(S) {
        let start = 0, n = S.length, len = 1;
        for (let i = 0; i < n; i++) {
            // odd length expandable
            let left = i - 1, right = i + 1;
            while (left >= 0 && right < n && S[left] === S[right]) {
                if (len < right - left + 1) {
                    len = right - left + 1;
                    start = left;
                }
                left--;
                right++;
            }

            // even length expandable
            left = i;
            right = i + 1;
            while (left >= 0 && right < n && S[left] === S[right]) {
                if (len < right - left + 1) {
                    len = right - left + 1;
                    start = left;
                }
                left--;
                right++;
            }
        }
        // both (start, end]
        return S.substring(start, start + len);
    }
}
```


Isomorphic Strings

Easy Accuracy: 34.21% Submissions: 115K+ Points: 2

Given two strings '**str1**' and '**str2**', check if these two strings are isomorphic to each other.

Two strings str1 and str2 are called isomorphic if there is a one to one mapping possible for every character of str1 to every character of str2 while **preserving the order**.

Note: All occurrences of every character in str1 should map to the same character in str2

Example 1:

Input:
str1 = aab
str2 = xxy
Output: 1
Explanation: There are two different charactersin aab and xxy, i.e a and b with frequency 2and 1 respectively.

Example 2:

Input:
str1 = aab
str2 = xyz
Output: 0
Explanation: There are two different charactersin aab but there are three different charactersin xyz. So there won't be one to one mapping between str1 and str2.

```cpp
class Solution {
    public:
    bool areIsomorphic(string s, string t) {
        int n = s.size(), m = t.size();
        if(n != m)
            return false;

        unordered_map<char, char> mp1, mp2;

        for (int i = 0; i < n; i++) {
            if (mp1.find(s[i]) != mp1.end() && mp1[s[i]] != t[i])
                return false;
            if (mp2.find(t[i]) != mp2.end() && mp2[t[i]] != s[i])
                return false;

            mp1[s[i]] = t[i];     // A to B
            mp2[t[i]] = s[i];     //  B to A
        }

        return true;
    }
};
```


1316. Distinct Echo Substrings

Hard 293 195 Add to List Share

Return the number of **distinct** non-empty substrings of `text` that can be written as the concatenation of some string with itself (i.e. it can be written as `a + a` where `a` is some string).

Example 1:

```
Input: text = "abcabcabc"
Output: 3
Explanation: The 3 substrings are "abcabc", "bcabca" and "cabcab".
```

Example 2:

```
Input: text = "leetcodeleetcode"
Output: 2
Explanation: The 2 substrings are "ee" and "leetcodeleetcode".
```


Constraints:

- `1 <= text.length <= 2000`
- `text` has only lowercase English letters.

```cpp
class Solution {
public:
    int distinctEchoSubstrings(string text) {
        int n = text.size();
        unordered_set<string> st;

        // trying every len possible strs
        for(int len=1; len<=n/2; len++) {
            int count =  0;

            for(int left=0, right=len; right<n; left++, right++) {
                if(text[left] == text[right]) {
                    count++;
                }
                else {
                    count = 0;
                }

                if(count==len) {
                    string possible = text.substr(left + 1, len);
                    if (st.find(possible) == st.end()) {
                        st.insert(possible);
                    }
                    // think it as we skipped that char
                    // abcabc match hogi par bcabca nahi and cout len nahi hoga
                    count--;
                }
            }
        }

        return st.size();
    }
};
```


Amazon 4 | Bolt 3 | Uber 2 | Adobe 2

290. Word Pattern

Easy | 6553 | 803 | Add to List | Share

Given a `pattern` and a string `s`, find if `s` follows the same pattern.

Here **follow** means a full match, such that there is a bijection between a letter in `pattern` and a **non-empty** word in `s`.

Example 1:

```
Input: pattern = "abba", s = "dog cat cat dog"
Output: true
```


Example 2:

```
Input: pattern = "abba", s = "dog cat cat fish"
Output: false
```

Example 3:

```
Input: pattern = "aaaa", s = "dog cat cat dog"
Output: false
```

```
/* If any of these two conditions hold, the answer will be "FALSE" ;
    #One character maps to two words
    #One word maps to two characters    */
vector<string> split(string s, char delimeter) {
    vector<string> group;
    int i = 0;
    string temp = "";
    while(i < s.size()){
        if(s[i] == delimeter){
            group.push_back(temp);
            temp = "";
        } else{
            temp += s[i];
        }
        i++;
    }
    group.push_back(temp);  // last wala

    return group;
}

bool wordPattern(string pattern, string s) {
    vector<string> words = split(s, ' ');
    if(words.size() != pattern.size()) return false;
    unordered_map<char, string> mp;
    set<string> used; // if we find one pattern with diff word again (duplicate)

    for(int i=0; i<pattern.size(); i++){
        if(mp.find(pattern[i]) != mp.end()){
            if(mp[pattern[i]] != words[i]) return false;
        }
        else{    // dog a ,dog c and c isnt there but dog is used
            if(used.count(words[i]) > 0) return false;
            mp[pattern[i]] = words[i];
            used.insert(words[i]);
        }
    }
    return true;
}
```



```
var wordPattern = function(pattern, s) {
    const v = s.split(' ');

    if (v.length !== pattern.length) {
        return false;
    }

    const mp = new Map();
    const st = new Set(); // for checking duplicate words

    for (i = 0; i < pattern.length; i++) {
        if (mp.has(pattern[i])) {
            if (mp.get(pattern[i]) !== v[i]) {
                return false;
            }
        } else {
            if (st.has(v[i])) {
                return false; // duplicate handling
                // a -> dog and then b -> dog
            }
            mp.set(pattern[i], v[i]);
            st.add(v[i]);
        }
    }
    return true;
};
```

Generate IP Addresses

Medium Accuracy: 38.71% Submissions: 33K+ Points: 4

Given a string **S** containing only digits, Your task is to complete the function **genIp()** which returns a vector containing all possible combinations of **valid** IPv4 IP addresses and takes only a string **S** as its only argument.

Note: Order doesn't matter. A **valid** IP address must be in the form of A.B.C.D, where A, B, C, and D are numbers from 0-255. The numbers cannot be 0 prefixed unless they are 0.

For string 11211 the IP address possible are

1.1.2.11

1.1.21.1

1.12.1.1

11.2.1.1

Example 1:

Input:
S = 1111
Output: 1.1.1.1

Example 2:

Input:
S = 55
Output: -1


```
class Solution{
  public:
    bool check(string str){
        int len = str.length();
        if(len == 0 || len > 3 || (str[0] == '0' && len > 1))
            return false;
        int p = stoi(str);
        if(p < 0 || p > 255) return false;
        return true;
    }
    string generate(int i, int j, int k, string s, int n){
        string A = s.substr(0, i+1);
        string B = s.substr(i+1, j-i);
        string C = s.substr(j+1, k-j);
        string D = s.substr(k+1, n-k-1);
        if(check(A) && check(B) && check(C) && check(D)){
            return A +"."+B +"."+C+ "."+D;
        }
        return "";
    }
    vector<string> genIp(string &s) {
        vector<string> v;
        int n = s.length();
        for(int i = 0; i < n; i++){
            for(int j = i + 1; j < n; j++){
                for(int k = j + 1; k < n; k++){
                    string st = generate( i, j, k, s, n );
                    if(st.length()){
                        v.push_back(st);
                    }
                }
            }
        }
        if(v.empty()){
            return {"-1"};
        }

        return v;
    }
};
```

```cpp
class Solution {
public:
    bool isValid(string s) {
        if (s.empty() || s.size() > 3 || (s.size() > 1 && s[0] == '0'))
            return false;
        int num = stoi(s);
        return num >= 0 && num <= 255;
    }

    void backtrack(string s, vector<string>& result, string current, int parts) {
        if (parts == 4 && s.empty()) {
            result.push_back(current);
            return;
        }

        for (int i = 1; i <= min(3, static_cast<int>(s.size())); i++) {
            string part = s.substr(0, i);
            if (isValid(part)) {
                string newS = s.substr(i);
                string newCurrent = current.empty() ? part : current + "." + part;
                backtrack(newS, result, newCurrent, parts + 1);
            }
        }
    }

    vector<string> genIp(string s) {
        vector<string> result;
        backtrack(s, result, "", 0);
        return result;
    }
};
```



```javascript
class Solution {
    check(str) {
        const len = str.length;
        if (len === 0 || len > 3 || (str[0] === '0' && len > 1))
            return false;
        const p = parseInt(str);
        if (p < 0 || p > 255)
            return false;
        return true;
    }
    generate(i, j, k, s, n) {
        const A = s.substring(0, i + 1);
        const B = s.substring(i + 1, j + 1);
        const C = s.substring(j + 1, k + 1);
        const D = s.substring(k + 1, n);
        if (this.check(A) && this.check(B) && this.check(C) && this.check(D)) {
            return `${A}.${B}.${C}.${D}`;
        }
        return "";
    }
    genIp(s) {
        const v = [];
        const n = s.length;
        for (let i = 0; i < n; i++) {
            for (let j = i + 1; j < n; j++) {
                for (let k = j + 1; k < n; k++) {
                    const st = this.generate(i, j, k, s, n);
                    if (st.length)
                        v.push(st);
                }
            }
        }
        if (v.length === 0)
            return ["-1"];

        return v;
    }
}
```

```javascript
class Solution {
    genIp2(s, idx, count, result, current) {
        if (count === 4 && idx === s.length) {
            current = current.slice(0, -1); // Remove the trailing dot
            result.push(current);
            return;
        }

        if (count === 4 || idx === s.length)
            return;

        for (let i = 1; i <= 3; i++) {
            if (idx + i > s.length) break;
            const part = s.substring(idx, idx + i);
            const val = parseInt(part);
            if (val > 255 || (i > 1 && s[idx] === '0')) {
                continue;
            }
            // if we come here that means its valid partition
            this.genIp2(s, idx + i, count + 1, result, current + part + '.');
        }
    }

    genIp(s) {
        const result = [];
        let current = '';
        this.genIp2(s, 0, 0, result, current);
        return result;
    }
}
```


Minimum characters to be added at front to make string palindrome

Hard Accuracy: **46.79%** Submissions: **31K+** Points: **8**

Given string **str** of length **N**. The task is to find the minimum characters to be added at the front to make string palindrome.

Note: A palindrome is a word which reads the same backward as forward. Example: "madam".

Example 1:

Input:
S = "abc"
Output: 2
Explanation:
Add 'b' and 'c' at front of above string to make it palindrome : "cbabc"

Example 2:

Input:
S = "aacecaaa"
Output: 1
Explanation: Add 'a' at front of above string to make it palindrome : "aaacecaaa"

Your Task:
You don't need to read input or print anything. Your task is to complete the function **minChar()** which takes a string S and returns an integer as output.

Expected Time Complexity: O(N)
Expected Auxiliary Space: O(N)

Constraints:
$1 <= S.length <= 10^6$

TLE

```
class Solution {
    public:
    int minChar(string str) {
        int left = 0, right = str.length() - 1;
        int endPoint = right;
        int res = 0;

        while(left < right){
            if(str[left] == str[right]){
                left++;
                right--;
            }
            else {
                res++;
                left = 0;
                right = --endPoint;
            }
        }

        return res;
    }
};
```

```
class Solution {
public:
/*   use KMP :
     nikalna whi he na ki last ke kitne element hata ke bachi hue
     string palindrome ban rhi he  to kmp algo agar lagade
     string +revere(string) me to to hume ans mil skta he
     kyu kyu? kyuki kmp algo se hum ye nikalte he ki larget
     preffix which is also suffix ...so nikal whi rhe          */

    int minChar(string str){
        int n = str.size();
        string copy = str;
        reverse(copy.begin(), copy.end());
        str += copy;

        int lps[2*n];                    constraints are 10 pow 6 so O(N) works
        lps[0] = 0;
        int i = 1;
        int j = 0;
        while(i < 2*n){
            if(str[i] == str[j]){
                lps[i] = j + 1;
                i++;
                j++;
            }
            else {
                if(j != 0)
                    j = lps[j-1];
                else{
                    lps[i] = 0;
                    i++;
                }
            }
        }

        return (n) - lps[2*n-1];
    }
};
```

```javascript
class Solution {
    minChar(str) {
        const n = str.length;
        const copy = str.split('').reverse().join('');
        str += copy;

        const lps = new Array(2 * n);
        lps[0] = 0;
        let i = 1;
        let j = 0;
        while (i < 2 * n) {
            if (str[i] === str[j]) {
                lps[i] = j + 1;
                i++;
                j++;
            } else {
                if (j !== 0)
                    j = lps[j - 1];
                else {
                    lps[i] = 0;
                    i++;
                }
            }
        }

        return n - lps[2 * n - 1];
    }
}

// Example usage
// const solution = new Solution();
// const str = "abacaba";
// const result = solution.minChar(str);
// console.log(result); // Outputs the minimum characters
```


Print Anagrams Together

Medium Accuracy: 65.78% Submissions: 58K+ Points: 4

Given an array of strings, return all groups of strings that are anagrams. The groups must be created in order of their appearance in the original array. Look at the sample case for clarification.

Note: The final output will be in lexicographic order.

Example 1:

Input:
N = 5
words[] = {act,god,cat,dog,tac}
Output:
act cat tac
god dog
Explanation:
There are 2 groups of
anagrams "god", "dog" make group 1.
"act", "cat", "tac" make group 2.

Example 2:

Input:
N = 3
words[] = {no,on,is}
Output:
is
no on
Explanation:
There are 2 groups of
anagrams "is" makes group 1.
"no", "on" make group 2.

```cpp
class Solution{
    public:
    vector<vector<string>> Anagrams(vector<string>& string_list) {
        unordered_map<string, vector<string>> mp;

        for(auto str: string_list) {
            string key = str;
            sort(key.begin(), key.end());
            mp[key].push_back(str);
        }

        vector<vector<string>> ans;
        for(auto vc: mp) {
            ans.push_back(vc.second);
        }

        return ans;
    }
};
```

```javascript
var groupAnagrams = function (strs) {
    const map = new Map()

    for (str of strs) {

        const sortStr = str.split('').sort().join()
        if (!map.has(sortStr)) {
            map.set(sortStr, [str])
        } else {
            map.get(sortStr).push(str)
        }

    }

    return Array.from(map.values())
};
```

```javascript
var groupAnagrams = function(strs) {
  let hash = {};

  for (let i = 0; i < strs.length; i++) {
    let key = strs[i].split('').sort();
    hash[key] = hash[key] || [];
    hash[key].push(strs[i]);

  }

  return Object.values(hash);
};
```

Smallest distinct window

Medium Accuracy: 31.85% Submissions: 79K+ Points: 4

Given a string '**s**'. The task is to find the **smallest** window length that contains all the characters of the given string at least one time.
For eg. A = **aabcbcdbca**, then the result would be 4 as of the smallest window will be **dbca**.

Example 1:

```
Input : "AABBBCBBAC"
Output : 3
Explanation : Sub-string -> "BAC"
```

Example 2:

```
Input : "aaab"
Output : 2
Explanation : Sub-string -> "ab"
```

Example 3:

```
Input : "GEEKSGEEKSFOR"
Output : 8
Explanation : Sub-string -> "GEEKSFOR"
```



```cpp
class Solution{
    public:
    int findSubString(string s) {
        int n = s.size();
        unordered_map<char, int> mp;
        for(int i=0; i<n; i++)
            mp[s[i]]++;

        int charCnt = mp.size();
        mp.clear();

        int left = 0, mnLen = INT_MAX;
        for(int right=0; right<n; right++) {
            mp[s[right]]++;
            if(mp[s[right]] == 1)
                charCnt--;
            while(charCnt == 0) {
                if(right - left + 1 < mnLen) {
                    mnLen = right - left + 1;
                }
                mp[s[left]]--;
                if(mp[s[left]] == 0)
                    charCnt++;

                left++;
            }
        }

        return mnLen;
    }
};
```

```javascript
class Solution {
    findSubString(s) {
        const n = s.length;
        const mp = new Map();
        for (let i = 0; i < n; i++) {
            mp.set(s[i], mp.get(s[i]) + 1 || 1);
        }

        let charCnt = mp.size;
        mp.clear();

        let left = 0;
        let mnLen = Infinity;
        for (let right = 0; right < n; right++) {
            if (!mp.has(s[right])) {
                mp.set(s[right], 0);
            }
            // mp[right]++;
            mp.set(s[right], mp.get(s[right]) + 1);
            if (mp.get(s[right]) === 1) {
                charCnt--;
            }
            while (charCnt === 0) {
                if (right - left + 1 < mnLen) {
                    mnLen = right - left + 1;
                }
                mp.set(s[left], mp.get(s[left]) - 1);
                if (mp.get(s[left]) === 0) {
                    charCnt++;
                }
                left++;
            }
        }

        return mnLen;
    }
}
```


Smallest window in a string containing all the characters of another string

Hard Accuracy: 30.19% Submissions: 113K+ Points: 8

Given two strings **S** and **P**. Find the smallest window in the string **S** consisting of all the characters(**including duplicates**) of the string **P**. Return "**-1**" in case there is no such window present. In case there are multiple such windows of same length, return the one with the least starting index.

Note : All characters are in Lowercase alphabets.

Example 1:

Input:
S = "timetopractice"
P = "toc"
Output:
toprac
Explanation: "toprac" is the smallest substring in which "toc" can be found.

Example 2:

Input:
S = "zoomlazapzo"
P = "oza"
Output:
apzo
Explanation: "apzo" is the smallest substring in which "oza" can be found.

```cpp
class Solution {
    public:
    string smallestWindow (string s, string p) {
        unordered_map<char, int> mp;
        for(int i=0; i<p.size(); i++)
            mp[p[i]]++;

        int leftChar = mp.size();  // there is no mp.empty()
        int len = INT_MAX;
        int start = 0, i = 0, j = 0;
        while(j < s.size()){
            mp[s[j]]--;
            // becz in aabc a is 2 times so decrease cnt if mp is 0
            if(mp[s[j]] == 0) leftChar--;
            while(leftChar == 0){
                if(j - i + 1 < len) {
                    len = j - i + 1;
                    start = i;
                }
                mp[s[i]]++;     // left window expanding
                if(mp[s[i]] == 1) leftChar++;
                i++;
            }
            j++;
        }

        if(len != INT_MAX) return s.substr(start, len);
        return "-1";
    }
};
```



```javascript
class Solution {
    smallestWindow(s, t) {
        const n = s.length;
        const m = new Map();

        for (const char of t) {
            m.set(char, m.get(char) + 1 || 1);
        }

        let i = 0, j = 0;
        let count = t.length;
        let ans = Infinity;
        let start = -1;

        while (j < n) {
            if (m.get(s[j]) > 0)
                count--;
            // fucking mp[x]--
            m.set(s[j], m.get(s[j]) - 1);
            // shrinking window
            while (count === 0) {
                if (j - i + 1 < ans) {
                    ans = j - i + 1;
                    start = i;
                }

                m.set(s[i], m.get(s[i]) + 1);
                if (m.get(s[i]) > 0) {
                    count++;
                }
                i++;
            }
            j++;
        }

        return start === -1 ? "-1" : s.substring(start, start + ans);
    }
}
```

Function to find Number of customers who could not get a computer

Write a function "runCustomerSimulation" that takes following two inputs

1. An integer 'n': total number of computers in a cafe and a string:
2. A sequence of uppercase letters 'seq': Letters in the sequence occur in pairs. The first occurrence indicates the arrival of a customer; the second indicates the departure of that same customer.

A customer will be serviced if there is an unoccupied computer. No letter will occur more than two times.
Customers who leave without using a computer always depart before customers who are currently using the computers. There are at most 20 computers per cafe.

For each set of input the function should output a number telling how many customers, if any walked away without using a computer. Return 0 if all the customers were able to use a computer.
runCustomerSimulation (2, "ABBAJJKZKZ") should return 0
runCustomerSimulation (3, "GACCBDDBAGEE") should return 1 as 'D' was not able to get any computer
runCustomerSimulation (3, "GACCBGDDBAEE") should return 0
runCustomerSimulation (1, "ABCBCA") should return 2 as 'B' and 'C' were not able to get any computer.
runCustomerSimulation(1, "ABCBCADEED") should return 3 as 'B', 'C' and 'E' were not able to get any computer.
Source: Fiberlink (maas360) Interview


```
int runCustomerSimulation(int n, const char *seq) {
    // seen[i] = 0, customer 'i' is not in cafe
    // seen[1] = 1, cust 'i' is in cafe but cmp isnt assigned yet.
    // seen[2] = 2, cust 'i' is in cafe and has occupied a computer.
    char seen[26] = {0};
    int res = 0;
    int occupied = 0;
    for (int i=0; seq[i]; i++) {
        int ind = seq[i] - 'A';
        if (seen[ind] == 0) {
            // set the current character as seen
            seen[ind] = 1;
            // assign a computer to new cust if enough
            if (occupied < n) {
                occupied++;
                // Set the current character as occupying a computer
                seen[ind] = 2;
            }
            else
                res++;
        }
        // If this is second occurrence of 'seq[i]'
        else { // Dec occupied only if this cust was using a computer
            if (seen[ind] == 2)
            occupied--;
            seen[ind] = 0;
        }
    }
    return res;
}
```

Word Break Leetcode sol

```
class Solution {
    wordBreak(A, B) {
        const dict = new Set(B);
        const n = A.length;
        const memo = new Array(n + 1).fill(-1);

        function solve(idx) {
            if (idx >= n) return 1;

            if (memo[idx] !== -1) return memo[idx];

            for (let len = 1; len <= n; len++) {
                const partition = A.substr(idx, len);
                if (dict.has(partition)) {
                    if (solve(idx + len)) return memo[idx] = 1;
                }
            }

            return memo[idx] = 0;
        }

        return solve(0);
    }
}
```


Longest Common Subsequence

Medium Accuracy: 41.68% Submissions: 210K+ Points: 4

Given two strings, find the length of longest subsequence present in both of them. Both the strings are in **uppercase** latin alphabets.

Example 1:

Input:
A = 6, B = 6
str1 = ABCDGH
str2 = AEDFHR
Output: 3
Explanation: LCS for input strings "ABCDGH" and "AEDFHR" is "ADH" of length 3.

Example 2:

Input:
A = 3, B = 2
str1 = ABC
str2 = AC
Output: 2
Explanation: LCS of "ABC" and "AC" is "AC" of length 2.

Reverse without data structure

```
class Solution{
public:
    void insertAtBottom(stack<int> &st, int num) {
        if(st.empty()) {
            st.push(num);
            return;
        }

        int save = st.top();
        st.pop();

        insertAtBottom(st, num);

        st.push(save);
    }
    void Reverse(stack<int> &st){

        if(st.empty()) return;

        int save = st.top();
        st.pop();

        Reverse(st);

        insertAtBottom(st, save);

    }
};
```

```
class Solution{
    public:
    int util(int i, int j, string &s1, string &s2,vector<vector<int>> &dp){
        if(i < 0 or j < 0)
            return 0;
        if(dp[i][j] != -1)
            return dp[i][j];
        if(s1[i] == s2[j])
            return dp[i][j] = 1 + util(i-1, j-1, s1, s2, dp);

        return dp[i][j] = max(util(i-1, j, s1, s2, dp), util(i, j-1, s1, s2, dp));
    }
    int lcs(int x, int y, string s1, string s2){
        vector<vector<int>> dp(x, vector<int>(y, -1));
        return util(x-1, y-1, s1, s2, dp);
    }
};
```

```
class Solution {
    public:
    int lcs(int n, int m, string s1, string s2) {
        vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
        for(int i=1; i<=n; i++) {
            for(int j=1; j<=m; j++){
                if(s1[i-1] == s2[j-1])
                    dp[i][j] = 1 + dp[i-1][j-1];
                else
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }
        return dp[n][m];
    }
};
```

Longest Repeating Subsequence

Medium Accuracy: 48.54% Submissions: 116K+ Points: 4

Given string str, find the length of the longest repeating subsequence such that it can be found twice in the given string.

The two identified subsequences A and B can use the same ith character from string str if and only if that ith character has different indices in A and B. For example, A = "xax" and B = "xax" then the index of first "x" must be different in the original string for A and B.

Example 1:

Input:
str = "axxzxy"
Output: 2
Explanation:
The given array with indexes looks like
a x x z x y
0 1 2 3 4 5

The longest subsequence is "xx".
It appears twice as explained below.

subsequence A
x x
0 1 <-- index of subsequence A
------
1 2 <-- index of str

subsequence B
x x
0 1 <-- index of subsequence B
------
2 4 <-- index of str

We are able to use character 'x'
(at index 2 in str) in both subsequences
as it appears on index 1 in subsequence A
and index 0 in subsequence B.


```cpp
class Solution {
public:
    int LongestRepeatingSubsequence(string s){
        int n = s.size();
        int dp[n+1][n+1];
        memset(dp, 0, sizeof(dp));

        for(int i=1; i<=n; i++) {
            for(int j=1; j<=n; j++) {
                if(i != j and s[i-1] == s[j-1]) {
                    dp[i][j] = 1 + dp[i-1][j-1];
                } else {
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
                }
            }
        }

        return dp[n][n];
    }
};
```



```javascript
/**
 * @param {string} str
 * @returns {number}
 */
class Solution {
    LongestRepeatingSubsequence(str) {
        const n = str.length;
        const dp = new Array(n + 1).fill(0).map(() => new Array(n + 1).fill(0));

        for (let i = 1; i <= n; i++) {
            for (let j = 1; j <= n; j++) {
                if (i !== j && str[i - 1] === str[j - 1]) {
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }

        return dp[n][n];
    }
}
```

Edit Distance

Medium Accuracy: 35.14% Submissions: 176K+ Points: 4

Given two strings **s** and **t**. Return the minimum number of operations required to convert **s** to **t**.

The possible operations are permitted:

1. Insert a character at any position of the string.
2. Remove any character from the string.
3. Replace any character from the string with any other character.

Example 1:

Input:
s = "geek", t = "gesek"
Output: 1
Explanation: One operation is required inserting 's' between two 'e's of s.

Example 2:

Input :
s = "gfg", t = "gfg"
Output:
0
Explanation: Both strings are same.


```cpp
class Solution {
    public:
    int m, n;
    int solve(int i, int j, string s, string t, vector<vector<int>> &dp) {
        if(i >= m) return dp[i][j] =  n-j;
        if(j >= n) return dp[i][j] = m-i;

        if(dp[i][j] != -1)
            return dp[i][j];

        if(s[i] == t[j])
            return dp[i][j] = solve(i+1, j+1, s, t, dp);
        else {
            int insert = 1 + solve(i, j+1, s, t, dp);
            int deletee = 1 + solve(i+1, j, s, t, dp);
            int change = 1 + solve(i+1, j+1, s, t, dp);

            return dp[i][j] = min({insert, deletee, change});
        }
    }
    int editDistance(string s, string t) {
        m = s.size(), n = t.size();
        vector<vector<int>> dp(m+1, vector<int> (n+1, -1));
        return solve(0, 0, s, t, dp);
    }
};
```



```cpp
class Solution {
public:
    int editDistance(string s, string t) {
        int m = s.size(), n = t.size();
        vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

        // Initialize the base cases
        for (int i = 0; i <= m; i++)
            dp[i][0] = i;
        for (int j = 0; j <= n; j++)
            dp[0][j] = j;

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (s[i - 1] == t[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1];
                } else {
                    dp[i][j] = 1 + min({dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]});
                }
            }
        }

        return dp[m][n];
    }
};
```

```javascript
class Solution {
    editDistance(s, t) {
        let m = s.length, n = t.length;
        let dp = new Array(m + 1).fill().map(() => new Array(n + 1).fill(0));

        for (let i = 0; i <= m; i++)
            dp[i][0] = i;

        for (let j = 0; j <= n; j++)
            dp[0][j] = j;

        for (let i = 1; i <= m; i++) {
            for (let j = 1; j <= n; j++) {
                if (s[i - 1] === t[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1];
                } else {
                    dp[i][j] = Math.min(dp[i - 1][j - 1], dp[i][j - 1], dp[i - 1][j]) + 1;
                }
            }
        }

        return dp[m][n];
    }
}
```


Remove Watermark Now

Transform String

Medium Accuracy: 29.76% Submissions: 33K+ Points: 4

Given two strings A and B. Find the minimum number of steps required to transform string A into string B. The only allowed operation for the transformation is selecting a character from string A and inserting it in the beginning of string A.

Example 1:

Input:
A = "abd"
B = "bad"
Output: 1
Explanation: The conversion can take place in 1 operation: Pick 'b' and place it at the front.

Example 2:

Input:
A = "GeeksForGeeks"
B = "ForGeeksGeeks"
Output: 3
Explanation: The conversion can take place in 3 operations:
Pick 'r' and place it at the front.
A = "rGeeksFoGeeks"
Pick 'o' and place it at the front.
A = "orGeeksFGeeks"
Pick 'F' and place it at the front.
A = "ForGeeksGeeks"

```cpp
class Solution {
public:
    int transform(string A, string B) {
        unordered_map<char, int> freqAB;
        for (char ch : A) freqAB[ch]++;
        for (char ch : B) freqAB[ch]--;

        for (auto it : freqAB) {
            if (it.second != 0)
                return -1;
        }

        int i = A.length() - 1;
        int j = B.length() - 1;
        int steps = 0;

        while (i >= 0 && j >= 0) {
            if (A[i] == B[j]) {
                i--, j--;
            } else {
                steps++;
                i--;
            }
        }

        return steps;
    }
};
```



```javascript
class Solution {
    transform(A, B) {
        const n = A.length, m = B.length;
        if (n !== m) return -1;

        const freq = new Array(256).fill(0);
        for (let i=0; i<n; i++) freq[A.charCodeAt(i)]++;
        for (let i=0; i<n; i++) freq[B.charCodeAt(i)]--;

        for (let i = 0; i < freq.length; i++) {
            if (freq[i] !== 0) return -1;
        }

        let i = n - 1, j = m - 1;
        let count = 0;
        // checking from last
        while (i >= 0) {
            if (A.charAt(i) !== B.charAt(j)) {
                count++;
                i--;
            } else {
                i--, j--;
            }
        }

        return count;
    }
}
```

1035. Uncrossed Lines

Medium 3569 48 Add to List Share

You are given two integer arrays `nums1` and `nums2`. We write the integers of `nums1` and `nums2` (in the order they are given) on two separate horizontal lines.

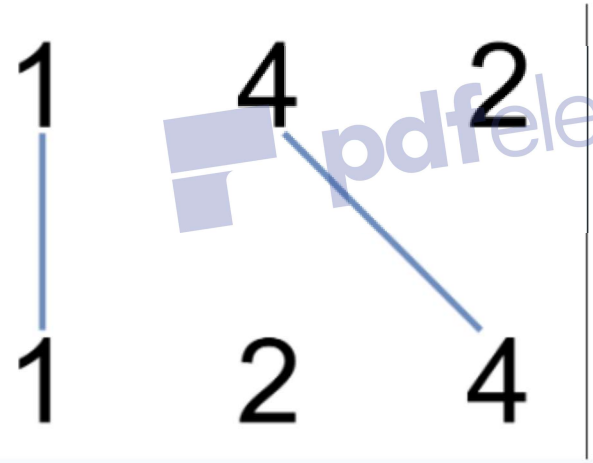
We may draw connecting lines: a straight line connecting two numbers `nums1[i]` and `nums2[j]` such that:

- `nums1[i] == nums2[j]`, and
- the line we draw does not intersect any other connecting (non-horizontal) line.

Note that a connecting line cannot intersect even at the endpoints (i.e., each number can only belong to one connecting line).

Return *the maximum number of connecting lines we can draw in this way.*

Example 1:




```
Input: nums1 = [1,4,2], nums2 = [1,2,4]
Output: 2
Explanation: We can draw 2 uncrossed lines as in the diagram.
We cannot draw 3 uncrossed lines, because the line from nums1[1] = 4 to nums2[2] = 4 will
intersect the line from nums1[2]=2 to nums2[1]=2.
```

Example 2:

```
Input: nums1 = [2,5,1,2,5], nums2 = [10,5,2,1,5,2]
Output: 3
```

```cpp
class Solution {
public:
    int maxUncrossedLines(vector<int>& nums1, vector<int>& nums2) {
        int m = nums1.size(), n = nums2.size();
        vector<vector<int>>dp (m+1, vector<int>(n+1, 0));

        for(int i=1; i<=m; i++){
            for(int j=1; j<=n; j++){
                if(nums1[i-1] == nums2[j-1])
                    dp[i][j] = 1 + dp[i-1][j-1];
                else
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }
        return dp[m][n];
    }
};
```

Wildcard string matching

Hard Accuracy: 23.88% Submissions: 24K+ Points: 8

Given two strings **wild** and **pattern** where wild string may contain wild card characters and pattern string is a normal string. Determine if the two strings match. The following are the allowed wild card characters in first string :-

* --> This character in string wild can be replaced by any sequence of characters, it can also be replaced by an empty string.
? --> This character in string wild can be replaced by any one character.

Example 1:

Input: wild = ge*ks
pattern = geeks
Output: Yes
Explanation: Replace the '*' in wild string with 'e' to obtain pattern "geeks".

Example 2:

Input: wild = ge?ks*
pattern = geeksforgeeks
Output: Yes
Explanation: Replace '?' and '*' in wild string with 'e' and 'forgeeks' respectively to obtain pattern "geeksforgeeks"

Your Task:
You don't need to read input or print anything. Your task is to complete the function **match()** which takes the string wild and pattern as input parameters and returns true if the string wild can be made equal to the string pattern, otherwise, returns false.

Expected Time Complexity: O(length of wild string * length of pattern string)
Expected Auxiliary Space: O(length of wild string * length of pattern string)

```cpp
class Solution {
public:
    int dp[1001][1001];
    bool solve(int i, int j, string& wild, string& pattern) {
        if (i == 0 && j == 0)
            return dp[i][j] = true;
        if (j == 0) {
            for (int k = 0; k < i; k++) {
                if (wild[k] != '*')
                    return dp[i][j] = false;
            }
            return dp[i][j] = true;
        }

        if (dp[i][j] != -1)
            return dp[i][j];

        if (wild[i-1] == pattern[j-1] || wild[i-1] == '?')
            return dp[i][j] = solve(i - 1, j - 1, wild, pattern);

        if (wild[i-1] == '*')
            return dp[i][j] = solve(i-1, j, wild, pattern) || solve(i, j-1, wild, pattern);

        return dp[i][j] = 0;
    }

    bool match(string wild, string pattern) {
        memset(dp, -1, sizeof(dp));
        int n = wild.size();
        int m = pattern.size();
        return solve(n, m, wild, pattern);
    }
};
```

```cpp
class Solution {
    public:
    bool match(string p, string s) {
        int n = s.length(), m = p.length();
        vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
        dp[0][0] = true;

        for(int j = 1; j <= m; j++){
            bool flag = true;
            for(int k = 1; k <= j; k++){
                if(p[k-1] != '*'){
                    flag = false;
                    break;
                }
            }
            dp[0][j] = flag;
        }

        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= m; j++){

                if(s[i-1] == p[j-1] || p[j-1] == '?')
                    dp[i][j] = dp[i-1][j-1];

                else if(p[j-1] == '*')
                    dp[i][j] = (dp[i-1][j] || dp[i][j-1]);

                else
                    dp[i][j] = false;
            }
        }

        return dp[n][m];
    }
};
```


Count Palindromic Subsequences

Medium Accuracy: 17.0% Submissions: 120K+ Points: 4

Given a string str of length N, you have to find number of palindromic subsequence (need not necessarily be distinct) present in the string str.
Note: You have to return the answer module 10^9+7;

Example 1:

Input:
Str = "abcd"
Output:
4
Explanation:
palindromic subsequence are : "a" ,"b", "c" ,"d"

Example 2:

Input:
Str = "aab"
Output:
4
Explanation:
palindromic subsequence are :"a", "a", "b", "aa"

```cpp
class Solution{
    public:
    long long mod = 1e9+7;
    long long int solve(string &str, int i, int j, vector<vector< long long int>> &dp){
        if(i > j)
            return dp[i][j] = 0;
        if(i == j)
            return dp[i][j] = 1;
        if(dp[i][j] != -1)
            return dp[i][j];

        if(str[i] == str[j])
          return dp[i][j] = (1 + solve(str, i+1, j, dp) +
                               solve(str, i, j-1, dp)) % mod;

        else
          return dp[i][j] = (mod + solve(str,i+1,j,dp) +
                            solve(str,i, j-1, dp) -
                            solve(str,i+1,j-1,dp)) % mod;    // +mod becz can go negative
    }

    long long int  countPS(string str){
       int n = str.length();
       vector<vector< long long int > > dp(n,vector< long long int>(n,-1));
       return solve(str,0,n-1,dp);
    }
};
```

```cpp
class Solution {
public:
    long long mod = 1000000007;
    long long int countPS(string str) {
        int n = str.size();
        vector<vector<long long int>> dp(n, vector<long long int>(n, 0));

        for (int i = 0; i < n; i++)
            dp[i][i] = 1;

        for (int len = 2; len <= n; len++) {
            for (int i = 0; i < n - len + 1; i++) {
                int j = i + len - 1;

                if (str[i] == str[j])
                    dp[i][j] = (1 + dp[i+1][j] + dp[i][j-1]) % mod;
                else
                    dp[i][j] = (dp[i+1][j] + dp[i][j-1] - dp[i+1][j-1] + mod) % mod;
            }
        }

        return dp[0][n - 1];
    }
};
```

```javascript
/** * @param {string} str
    @returns {number}   */
class Solution {
    countPS(str) {
        let mod = 1000000007;
        let n = str.length;
        let dp = new Array(n).fill(0).map(() => new Array(n).fill(0));

        for (let i = 0; i < n; i++) {
            dp[i][i] = 1;
        }

        for (let len = 2; len <= n; len++) {
            for (let i = 0; i < n - len + 1; i++) {
                let j = i + len - 1;

                if (str[i] === str[j]) {
                    dp[i][j] = (1 + dp[i + 1][j] + dp[i][j - 1]) % mod;
                } else {
                    dp[i][j] = (dp[i + 1][j] + dp[i][j - 1] - dp[i + 1][j - 1] + mod) % mod;
                }
            }
        }

        return dp[0][n - 1];
    }
}
```

745. Prefix and Suffix Search

Hard 2201 472 Add to List Share

Design a special dictionary that searches the words in it by a prefix and a suffix.

Implement the `WordFilter` class:

- `WordFilter(string[] words)` Initializes the object with the `words` in the dictionary.
- `f(string pref, string suff)` Returns *the index of the word in the dictionary,* which has the prefix `pref` and the suffix `suff`. If there is more than one valid index, return **the largest** of them. If there is no such word in the dictionary, return `-1`.

Example 1:

```
Input
["WordFilter", "f"]
[[["apple"]], ["a", "e"]]
Output
[null, 0]
Explanation
WordFilter wordFilter = new WordFilter(["apple"]);
wordFilter.f("a", "e"); // return 0, because the word at index
0 has prefix = "a" and suffix = "e".
```



```cpp
class WordFilter {
public:
    unordered_map<string,int>mp;
    // word.substr(0,i+1) + '|' + word.substr(j)
    WordFilter(vector<string>& words) {

        for(int idx = 0; idx < words.size(); idx++)
        { // prefix+suffix   with dummy
            string word = words[idx];
            int l = word.size();
            for(int i = 0; i<l; i++)
            {
                for(int j = l-1; j>=0; j--)
                {
// map me sare pref suff combs with idx store (a e, a le, a ple....ap e, ap le and so on)
                    string prefix = word.substr(0,i+1);
                    string suffix = word.substr(j, l);
                    mp[prefix + '|' + suffix] = idx;      // k becz index return karna hai
                }
            }
        }
    }

    int f(string prefix, string suffix) {
        if(mp.find(prefix + '|' + suffix) == mp.end()) return -1;
        return mp[prefix + '|' + suffix];   // return idx that is stored in map
    }
};
```

```cpp
class WordFilter {
   private:
    unordered_map<string, int> hashMap;
   public:
    WordFilter(vector<string>& words) {
        int n = words.size();

        for (int i = 0; i < n; i++) {
            string word = words[i];
            int wordSize = word.size();

            for (int j = 0; j <= wordSize; j++) {
                string pref = word.substr(0, j);  // a , ap, app ..
                for (int k = 0; k <= wordSize; k++) {
                    string suff = word.substr(k);    // apple, pple ...
                    hashMap[pref + "|" + suff] = i + 1;
                }
            }
        }
    }

    int f(string prefix, string suffix) {
        string s = prefix + "|" + suffix;
        return hashMap[s] - 1;
    }
};
```



```javascript
/** @param {string[]} words */
var WordFilter = function (words) {
  this.map = new Map();
  for (let i = 0; i < words.length; i++) {
    let prefix = '';
    for (let j = 0; j <= words[i].length; j++) {
      let suffix = '';
      for (let k = 0; k <= words[i].length; k++) {
        suffix = words[i].slice(k);
        this.map.set(prefix + '#' + suffix, i);
      }
      prefix += words[i][j];
    }
  }
}

WordFilter.prototype.f = function (prefix, suffix) {
  let target = prefix + '#' + suffix;
  return this.map.has(target) ? this.map.get(target) : -1;
}

/**
 * Your WordFilter object will be instantiated and called as such:
 * var obj = new WordFilter(words);
 * var param_1 = obj.f(pref, suff);
 */
```

68. Text Justification

Hard 2451 3577 Add to List Share

Given an array of strings `words` and a width `maxWidth`, format the text such that each line has exactly `maxWidth` characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces `' '` when necessary so that each line has exactly `maxWidth` characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line does not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left-justified, and no extra space is inserted between words.

Note:

- A word is defined as a character sequence consisting of non-space characters only.
- Each word's length is guaranteed to be greater than `0` and not exceed `maxWidth`.
- The input array `words` contains at least one word.

Example 1:

```
Input: words = ["This", "is", "an", "example", "of", "text",
"justification."], maxWidth = 16
Output:
[
   "This    is    an",
   "example  of text",
   "justification.  "
]
```

Example 2:

```
Input: words = ["What","must","be","acknowledgment","shall","be"], maxWidth = 16
Output:
[
  "What   must   be",
  "acknowledgment  ",
  "shall be        "
]
Explanation: Note that the last line is "shall be    " instead of "shall     be", because the last line must be left-justified instead
of fully-justified.
Note that the second line is also left-justified because it contains only one word.
```

Example 3:

```
Input: words =
["Science","is","what","we","understand","well","enough","to","explain","to","a","computer.","Art","is","everything","else","we","do"],
maxWidth = 20
Output:
[
  "Science  is  what we",
  "understand      well",
  "enough to explain to",
  "a  computer.  Art is",
  "everything  else  we",
  "do                  "
]
```

```
public:
    vector<string> fullJustify(vector<string>& words, int maxWidth) {
        vector<string> result;
        vector<string> currLine;
        int currWidth = 0;
        for (int i = 0; i < words.size(); i++) {

            // Calculate currWidth and see if this line can accomodate currWord
            int value = currLine.size() == 0 ? words[i].length() : (currWidth + words[i].length() + 1);

            if (value <= maxWidth) {
                currLine.push_back(words[i]);
                currWidth = value;
            } else {

                if (currLine.size() > 1) {
                    processLineWithKWords(currLine, currWidth, maxWidth, result);
                } else {
                    processLineWithOneWord(currLine, currWidth, maxWidth, result);
                }

                currLine.clear();
                currLine.push_back(words[i]);
                currWidth = words[i].length();
            }
        }

        processLastLine(currLine, currWidth, maxWidth, result);

        return result;
    }
};

class Solution {
public:
    void processLineWithKWords(vector<string>& currLine, int currWidth, int maxWidth, vector<string>& result) {
        int whiteSpaces = maxWidth - currWidth;
        int evenSpaces = whiteSpaces / (currLine.size() - 1);
        int unevenSpaces = whiteSpaces % (currLine.size() - 1);

        int word = 0;
        // adding even spaces / wala
        while (word < currLine.size() - 1) {
            int spaces = 0;
            while (spaces < evenSpaces) {
                currLine[word] += " ";
                spaces++;
            }
            word++;
        }

        // Left must have extra spaces left side me wo bhi
        word = 0;
        while (unevenSpaces > 0) {
            currLine[word] += " ";
            word++;
            unevenSpaces--;
        }

        string tempRes = "";
        for (int k = 0; k < currLine.size(); k++) {
            if (k == 0) {
                tempRes += currLine[k];
            } else {
                tempRes = tempRes + " " + currLine[k];
            }
        }

        result.push_back(tempRes);
    }

    void processLineWithOneWord(vector<string>& currLine, int currWidth, int maxWidth, vector<string>& result) {
        int whiteSpaces = maxWidth - currWidth;
        string tempRes = currLine[0];
        while (whiteSpaces > 0) {
            tempRes += " ";
            whiteSpaces--;
        }
        result.push_back(tempRes);
    }

    void processLastLine(vector<string>& currLine, int currWidth, int maxWidth, vector<string>& result) {
        int whiteSpaces = maxWidth - currWidth;
        string tempRes = currLine[0];

        if (currLine.size() > 1) {
            for (int i = 1; i < currLine.size(); i++) {
                tempRes += " " + currLine[i];
            }
        }

        while (whiteSpaces > 0) {
            tempRes += " ";
            whiteSpaces--;
        }

        result.push_back(tempRes);
    }
```

```javascript
/* @param {string[]} words
 * @param {number} maxWidth
 * @return {string[]}    */
var fullJustify = function(words, maxWidth) {
    function processLineWithKWords(currLine, currWidth, maxWidth, result) {
        let whiteSpaces = maxWidth - currWidth;
        let evenSpaces = Math.floor(whiteSpaces / (currLine.length - 1));
        let unevenSpaces = whiteSpaces % (currLine.length - 1);

        let word = 0;
        // adding even spaces / wala
        while (word < currLine.length - 1) {
            let spaces = 0;
            while (spaces < evenSpaces) {
                currLine[word] += " ";
                spaces++;
            }
            word++;
        }

        // Left must have extra spaces left side me wo bhi
        word = 0;
        while (unevenSpaces > 0) {
            currLine[word] += " ";
            word++;
            unevenSpaces--;
        }

        let tempRes = currLine.join(" ");
        result.push(tempRes);
    }

    function processLineWithOneWord(currLine, currWidth, maxWidth, result) {
        let whiteSpaces = maxWidth - currWidth;
        let tempRes = currLine[0];
        while (whiteSpaces > 0) {
            tempRes += " ";
            whiteSpaces--;
        }
        result.push(tempRes);
    }

    function processLastLine(currLine, currWidth, maxWidth, result) {
        let whiteSpaces = maxWidth - currWidth;
        let tempRes = currLine.join(" ");

        while (whiteSpaces > 0) {
            tempRes += " ";
            whiteSpaces--;
        }

        result.push(tempRes);
    }

    let result = [];
    let currLine = [];
    let currWidth = 0;

    for (let i = 0; i < words.length; i++) {
        let value = currLine.length === 0 ? words[i].length : currWidth + words[i].length + 1;

        if (value <= maxWidth) {
            currLine.push(words[i]);
            currWidth = value;
        } else {
            if (currLine.length > 1) {
                processLineWithKWords(currLine, currWidth, maxWidth, result);
            } else {
                processLineWithOneWord(currLine, currWidth, maxWidth, result);
            }

            currLine = [words[i]];
            currWidth = words[i].length;
        }
    }

    processLastLine(currLine, currWidth, maxWidth, result);

    return result;
};
```

Transform String

Medium Accuracy: 29.76% Submissions: 33K+ Points: 4

Given two strings A and B. Find the minimum number of steps required to transform string A into string B. The only allowed operation for the transformation is selecting a character from string A and inserting it in the beginning of string A.

Example 1:

Input:
A = "abd"
B = "bad"
Output: 1
Explanation: The conversion can take place in 1 operation: Pick 'b' and place it at the front.

Example 2:

Input:
A = "GeeksForGeeks"
B = "ForGeeksGeeks"
Output: 3
Explanation: The conversion can take place in 3 operations:
Pick 'r' and place it at the front.
A = "rGeeksFoGeeks"
Pick 'o' and place it at the front.
A = "orGeeksFGeeks"
Pick 'F' and place it at the front.
A = "ForGeeksGeeks"


```cpp
class Solution {
public:
    int transform(string A, string B) {
        unordered_map<char, int> freqAB;
        for (char ch : A) freqAB[ch]++;
        for (char ch : B) freqAB[ch]--;

        // if diff elm or freq return -1
        for (auto it : freqAB) {
            if (it.second != 0)
                return -1;
        }

        int i = A.length() - 1;
        int j = B.length() - 1;
        int steps = 0;

        // since freq is same we need to match order so
        // we start from last if mismatch cnt++ ie shfted
        while (i >= 0 && j >= 0) {
            if (A[i] == B[j]) {
                i--, j--;
            } else {
                steps++;
                i--;
            }
        }

        return steps;
    }
};
```

Word Wrap

Hard Accuracy: 29.74% Submissions: 36K+ Points: 8

Given an array **nums[]** of size **n**, where **nums[i]** denotes the number of characters in one word. Let **K** be the limit on the number of characters that can be put in one line (line width). Put line breaks in the given sequence such that the lines are printed neatly.

Assume that the length of each word is smaller than the line width. When line breaks are inserted there is a possibility that extra spaces are present in each line. The extra spaces include spaces put at the end of every line **except the last one**.

You have to **minimize** the following total cost where **total cost** = Sum of cost of all lines, where cost of line is = (Number of extra spaces in the line)^2.

Example 1:

Input: nums = {3,2,2,5}, k = 6
Output: 10
Explanation: Given a line can have 6 characters,
Line number 1: From word no. 1 to 1
Line number 2: From word no. 2 to 3
Line number 3: From word no. 4 to 4
So total cost = $(6-3)^2 + (6-2-2-1)^2 = 3^2+1^2 = 10$.
As in the first line word length = 3 thus extra spaces = 6 - 3 = 3 and in the second line there are two word of length 2 and there already 1 space between two word thus extra spaces = 6 - 2 -2 -1 = 1. As mentioned in the problem description there will be no extra spaces in the last line. Placing first and second word in first line and third word on second line would take a cost of $0^2 + 4^2 = 16$ (zero spaces on first line and 6-2 = 4 spaces on second), which isn't the minimum possible cost.

Example 2:

Input: nums = {3,2,2}, k = 4
Output: 5
Explanation: Given a line can have 4 characters,
Line number 1: From word no. 1 to 1
Line number 2: From word no. 2 to 2
Line number 3: From word no. 3 to 3
Same explaination as above total cost = $(4 - 3)^2 + (4 - 2)^2 = 5$.

```cpp
class Solution {
public:
    int dp[500][2000];
    int solve(int ind, vector<int> &nums, int remSpaces, int k) {
        if(ind >= nums.size())
            return 0;
        if(dp[ind][remSpaces] != -1)
            return dp[ind][remSpaces];

        int ans;
        // If space is not there then we have to keep the
        // word in the next line
        if(nums[ind] > remSpaces) {
            ans = (remSpaces+1)*(remSpaces+1) +
            solve(ind+1, nums, k-nums[ind]-1, k);
        }
        // else we have two cases either to include the word
        // in the same or not to include in the same line
        else {
            int includeInThisLine = solve(ind+1, nums, remSpaces-nums[ind]-1, k);

            int excludeFromThisLine = (remSpaces+1)*(remSpaces+1) +
                                      solve(ind+1, nums, k-nums[ind]-1, k);

            ans = min(includeInThisLine, excludeFromThisLine);
        }

        return dp[ind][remSpaces] = ans;
    }

    int solveWordWrap(vector<int>nums, int k) {
        memset(dp, -1, sizeof(dp));
        return solve(0, nums, k, k);
    }
};
```

```javascript
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number}
 */
class Solution {
    constructor() {
        this.dp = new Array(500).fill().map(() => new Array(2000).fill(-1));
    }

    solve(ind, nums, remSpaces, k) {
        if (ind >= nums.length)
            return 0;
        if (this.dp[ind][remSpaces] !== -1)
            return this.dp[ind][remSpaces];

        let ans;
        if (nums[ind] > remSpaces) {
            ans = (remSpaces + 1) * (remSpaces + 1) +
                  this.solve(ind + 1, nums, k - nums[ind] - 1, k);
        } else {
            let includeInThisLine = this.solve(ind + 1, nums, remSpaces - nums[ind] - 1, k);

            let excludeFromThisLine = (remSpaces + 1) * (remSpaces + 1) +
                                      this.solve(ind + 1, nums, k - nums[ind] - 1, k);

            ans = Math.min(includeInThisLine, excludeFromThisLine);
        }

        return this.dp[ind][remSpaces] = ans;
    }

    solveWordWrap(nums, k) {
        return this.solve(0, nums, k, k);
    }
}
```


468. Validate IP Address

Medium 897 2626 Add to List Share

Given a string `queryIP`, return `"IPv4"` if IP is a valid IPv4 address, `"IPv6"` if IP is a valid IPv6 address or `"Neither"` if IP is not a correct IP of any type.

A valid IPv4 address is an IP in the form `"$x_1$.$x_2$.$x_3$.$x_4$"` where `0 <= $x_i$ <= 255` and `$x_i$` **cannot contain** leading zeros. For example, `"192.168.1.1"` and `"192.168.1.0"` are valid IPv4 addresses while `"192.168.01.1"`, `"192.168.1.00"`, and `"192.168@1.1"` are invalid IPv4 addresses.

A valid IPv6 address is an IP in the form `"$x_1$:$x_2$:$x_3$:$x_4$:$x_5$:$x_6$:$x_7$:$x_8$"` where:

- `1 <= $x_i$.length <= 4`
- `$x_i$` is a **hexadecimal string** which may contain digits, lowercase English letter (`'a'` to `'f'`) and upper-case English letters (`'A'` to `'F'`).
- Leading zeros are allowed in `$x_i$`.

For example, "`2001:0db8:85a3:0000:0000:8a2e:0370:7334`" and "`2001:db8:85a3:0:0:8A2E:0370:7334`" are valid IPv6 addresses, while "`2001:0db8:85a3::8A2E:037j:7334`" and "`02001:0db8:85a3:0000:0000:8a2e:0370:7334`" are invalid IPv6 addresses.

Example 1:

```
Input: queryIP = "172.16.254.1"
Output: "IPv4"
Explanation: This is a valid IPv4 address, return "IPv4".
```

Example 2:

```
Input: queryIP = "2001:0db8:85a3:0:0:8A2E:0370:7334"
Output: "IPv6"
Explanation: This is a valid IPv6 address, return "IPv6".
```

Example 3:

```
Input: queryIP = "256.256.256.256"
Output: "Neither"
Explanation: This is neither a IPv4 address nor a IPv6 address.
```

```cpp
string validIPAddress(string queryIP) {
    int dot = 0, colon = 0;
    for (char ch : queryIP) {
        if (ch == '.') dot += 1;
        if (ch == ':') colon += 1;
    }
    if (dot != 3 && colon != 7) return "Neither";

    if (dot > 0) {
        vector<string> ips = split(queryIP, '.');
        if (ips.size() != 4) return "Neither";

        for (int i = 0; i < ips.size(); i++) {
            if (ips[i].length() > 1 && ips[i][0] == '0')
                return "Neither";   // leading zeroes at beginning

            for (char ch: ips[i]) {
                if (!isdigit(ch))
                    return "Neither";
            }

            if (ips[i].size()==0 || ips[i].size()>3 || stoi(size[i])<0 || stoi(ips[i])>255) {
                return "Neither";
            }

        }
        return "IPv4";

    } else {
        vector<string> ips = split(queryIP, ':');
        if (ips.size() != 8) return "Neither";

        for (int i = 0; i < ips.size(); i++) {
            if (ips[i].length() > 4 || ips[i].length() == 0)
                return "Neither";

            for (char ch : ips[i]) {
                if (!isdigit(ch) && (ch < 'a' || ch > 'f') && (ch < 'A' || ch > 'F'))
                    return "Neither";
            }
        }

        return "IPv6";
    }
}
```

```cpp
vector<string> split(string s, char delimiter) {
    vector<string> parts;
    int i = 0, n = s.size();
    string part = "";
    while (i < n) {
        if(s[i] != delimiter)
            part += s[i];
        else {
            parts.push_back(part);
            part = "";
        }
        i++;
    }
    parts.push_back(part);
    return parts;
}
```

Sol. JS

```javascript
var validIPAddress = function(queryIP) {
    let dot = 0, colon = 0;
    for (let ch of queryIP) {
        if (ch === '.') dot++;
        if (ch === ':') colon++;
    }
    if (dot !== 3 && colon !== 7) return "Neither";
    if (dot > 0) {
        let ips = queryIP.split('.');
        if (ips.length !== 4) return "Neither";

        for (let i = 0; i < ips.length; i++) {
            if (ips[i].length > 1 && ips[i][0] === '0')
                return "Neither";   // leading zeroes at beginning

            for (let ch of ips[i]) {
                if (!isNaN(ch) && parseInt(ch) >= 0 && parseInt(ch) <= 9)  continue;
                else return "Neither";
            }
            if (ips[i].length === 0 || parseInt(ips[i]) < 0 || parseInt(ips[i]) > 255)
                return "Neither";
        }
        return "IPv4";

    } else {    // ipv6
        let ips = queryIP.split(':');
        if (ips.length !== 8) return "Neither";

        for (let i = 0; i < ips.length; i++) {
            if (ips[i].length > 4 || ips[i].length === 0) return "Neither";

            for (let ch of ips[i]) {
                if ((ch >= 'a' && ch <= 'f') || (ch >= 'A' && ch <= 'F') || (!isNaN(ch) && parseInt(ch) >= 0 && parseInt(ch) <= 9))
                    continue;
                else return "Neither";
            }
        }

        return "IPv6";
    }
};
```

763. Partition Labels

Medium 9722 356 Add to List Share

You are given a string `s`. We want to partition the string into as many parts as possible so that each letter appears in at most one part.

Note that the partition is done so that after concatenating all the parts in order, the resultant string should be `s`.

Return *a list of integers representing the size of these parts.*

Example 1:

```
Input: s = "ababcbacadefegdehijhklij"
Output: [9,7,8]
Explanation:
The partition is "ababcbaca", "defegde", "hijhklij".
This is a partition so that each letter appears in at most one part.
A partition like "ababcbacadefegde", "hijhklij" is incorrect, because
it splits s into less parts.
```

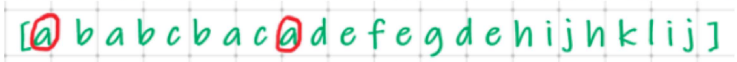
Example 2:

```
Input: s = "eccbbbbdec"
Output: [10]
```


- **Fundamental idea** is that **if two instances of characters** should be **in the same partition.**
- So, **we start with the first character** and **see at what point we can finish the partition.**
- The earliest we can do is at the **last instance of this character.**
- **What if we find a character between the first character and the last instance of it?**
  - In this case, **we increase the length of the partition**. Why?
  - **If we do not increase the partition,** the **new character ends up into two partitions**, which violates the constraint of the problem.
- If we have **gone through all the characters between the start of partition and maximum of the last instance of characters**, we **can close the partition and start a new one.**
- **Runtime: O(n)**. We analyze **each character twice.**
- **Memory: O(1).** We **use 26 elements for the algorithm**. Additional memory is used to store the result; the algorithm does not need it.
- DRY RUN TO A TEST CASE AT THE END OF CODE.

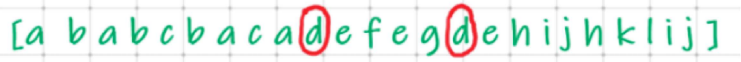
```cpp
class Solution {
public:
    vector<int> partitionLabels(string s) {
        // vector for keeping the track of  last occurence of every character
        vector<int> end_idx(26,0);

        for(int i = 0; i < s.length(); ++i)
            end_idx[s[i] - 'a']  = i;

        vector<int> res;

        int start = 0, end = 0;
        // scanning string character by character
        for(int i = 0; i < s.length(); ++i)
        {
        // whenever we get an character we check,
        // last index of that character
            end = max(end, end_idx[s[i] - 'a']);

            // when current i.e i == end
            // add it to result
            if( i == end)
            {
                // all the characters of current partition included
                res.push_back(i - start + 1);
                // update the start pointer for fresh start
                start = i + 1;
            }
        }

        return res;
    }
};
```

Let's take **an example and see how it works**. **Start with the first character of the string** and **find the last instance of that character**, *in this case, it is a*

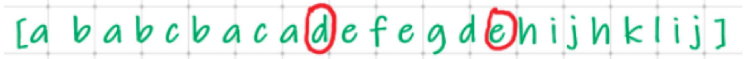


Go through all the characters *until the last instance*, **if the last instance of any of the characters is greater than the current last instance**.

- *expand the partition to include new last instance*. **In this example**, **none of the characters have the last index beyond the last index of a.**
- **Once, we reach the last index**, we can **close the current partition and start a new one**. Find the **last instance of the first character of the new partition.**



Go though the characters in between. In this example, **character e has last instance greater than current one**, so we **update the partition till e**.



There is **no character in between with the last instance greater than the last instance of e**, so we **close the partition** there and **start the next one from the next character in the string which is h**

Next character **i will expand the partition to index 22** and next to next character j will push it to 23. There is no other character that has the last occurrence after that index. Hence, we would close the partition there.

Whenever we are closing the partition, we would **add the length of that partition end – start + 1 in a list to return.**

942. DI String Match

Easy 2275 914 Add to List Share

A permutation `perm` of `n + 1` integers of all the integers in the range `[0, n]` can be represented as a string `s` of length `n` where:

- `s[i] == 'I'` if `perm[i] < perm[i + 1]`, and
- `s[i] == 'D'` if `perm[i] > perm[i + 1]`.

Given a string `s`, reconstruct the permutation `perm` and return it. If there are multiple valid permutations perm, return **any of them**.

Example 1:

```
Input: s = "IDID"
Output: [0,4,1,3,2]
```

Example 2:

```
Input: s = "III"
Output: [0,1,2,3]
```

Example 3:

```
Input: s = "DDI"
Output: [3,2,0,1]
```



```cpp
class Solution {
public:
    vector diStringMatch(string s) {
        vector v;
        int n = s.length();
        int l = 0 , h = n;
        for(int i = 0 ; i < n ; i++) {
            if(s[i] == 'I' ){
                v.push_back(l);
                l++;
            }
            else if (s[i] == 'D') {
                v.push_back(h);
                h--;
            }
        }

        if(s[n-1] == 'I')
            v.push_back(l);
        else if(s[n-1] == 'D')
            v.push_back(h);

        return v;
    }
};
```

821. Shortest Distance to a Character

Easy 2934 150 Add to List Share

Given a string `s` and a character `c` that occurs in `s`, return *an array of integers* `answer` *where* `answer.length == s.length` *and* `answer[i]` *is the* ***distance*** *from index* `i` *to the* ***closest*** *occurrence of character* `c` *in* `s`.

The **distance** between two indices `i` and `j` is `abs(i - j)`, where `abs` is the absolute value function.

Example 1:

```
Input: s = "loveleetcode", c = "e"
Output: [3,2,1,0,1,0,0,1,2,2,1,0]
Explanation: The character 'e' appears at indices 3, 5, 6, and 11 (0-indexed).
The closest occurrence of 'e' for index 0 is at index 3, so the distance is abs(0 - 3) = 3.
The closest occurrence of 'e' for index 1 is at index 3, so the distance is abs(1 - 3) = 2.
For index 4, there is a tie between the 'e' at index 3 and the 'e' at index 5, but the distance is still the same: abs(4 - 3) == abs(4 - 5) = 1.
The closest occurrence of 'e' for index 8 is at index 6, so the distance is abs(8 - 6) = 2.
```

Example 2:

```
Input: s = "aaab", c = "b"
Output: [3,2,1,0]
```

```cpp
class Solution {
public:
    vector<int> shortestToChar(string s, char c) {
        vector<int>out;
        vector<int>pos;

        //collect all the position of given char
        for(int i=0; i<s.size(); i++) {
            if(s[i]==c)
                pos.push_back(i);
        }

//traversal and find the min diffrence from the given char to all given pos
        for(int i=0;i<s.size();i++) {
            int min_dis=INT_MAX;
            for(int j=0; j<pos.size(); j++) {
                min_dis = min(min_dis, abs(i-pos[j]));
            }
            out.push_back(min_dis);
        }

        return out;
    }
};
```

696. Count Binary Substrings

Easy 3741 806 Add to List Share

Given a binary string `s`, return the number of non-empty substrings that have the same number of `0`'s and `1`'s, and all the `0`'s and all the `1`'s in these substrings are grouped consecutively.

Substrings that occur multiple times are counted the number of times they occur.

Example 1:

```
Input: s = "00110011"
Output: 6
Explanation: There are 6 substrings that have equal number of consecutive 1's and 0's: "0011", "01", "1100", "10", "0011", and "01".
Notice that some of these substrings repeat and are counted the number of times they occur.
Also, "00110011" is not a valid substring because all the 0's (and 1's) are not grouped together.
```

Example 2:

```
Input: s = "10101"
Output: 4
Explanation: There are 4 substrings: "10", "01", "10", "01" that have equal number of consecutive 1's and 0's.
```


We use **curLen** to keep track of the count of continuous 0's or 1's we have seen. If we encounter a different digit, we make curLen **prevLen** and start a new counting cycle for the new digit in **curLen**.

The number of **prevLen** is saying that *"this is the maximum number of the opposite digit I can offer you to make a valid substring."*

So we can get the answer using this in linear time O(N).
For example, with "001110" at index 2,

```
"001110"
   i
prevLen = 2
curLen = 1
```

The previous part can, at most, offer two 0's to form a valid substring. So cnt++.

At index 3,

```
"001110"
    i
prevLen = 2
curLen = 2
```

The previous part can match two 0's with the two 1's we have seen. So cnt++.

At index 4,

```
"001110"
     i
prevLen = 2
curLen = 3
```

The previous part cannot match three 0's for the three 1's we have seen. So we don't do cnt++.

When we move to index 5, we store 3 in **prevLen** and start a new counting cycle for 0's in **curLen**, and keep doing this until the end.


```cpp
int countBinarySubstrings(string s) {
    int curLen = 1, prevLen = 0, cnt = 0;
    for (int i = 1; i < s.size(); i++) {
        if (s[i] == s[i - 1]) curLen++;
        else {
            prevLen = curLen;
            curLen = 1;
        }
        if (prevLen >= curLen) cnt++;
    }
    return cnt;
}
```

917. Reverse Only Letters

Easy 2002 65 Add to List Share

Given a string `s`, reverse the string according to the following rules:

- All the characters that are not English letters remain in the same position.
- All the English letters (lowercase or uppercase) should be reversed.

Return `s` *after reversing it.*

Example 1:

```
Input: s = "ab-cd"
Output: "dc-ba"
```

Example 2:

```
Input: s = "a-bC-dEf-ghIj"
Output: "j-Ih-gfE-dCba"
```

Example 3:

```
Input: s = "Test1ng-Leet=code-Q!"
Output: "Qedo1ct-eeLg=ntse-T!"
```



```cpp
class Solution {
public:
    string reverseOnlyLetters(string s) {
        int l = 0, r = s.size() - 1;
        while (l < r) {
            while (l < r && !isalpha(s[l])) ++l; // Skip non-alpha characters
            while (l < r && !isalpha(s[r])) --r; // Skip non-alpha characters
            swap(s[l++], s[r--]);
        }
        return s;
    }
};
```

```cpp
class Solution {
public:
    string reverseOnlyLetters(string s) {
        stack<char> letterStack;
        for (char c : s)
            if (isalpha(c))
                letterStack.push(c);

        for (char &c : s) {
            if (isalpha(c)) {
                c = letterStack.top();
                letterStack.pop();
            }
        }
        return s;
    }
};
```

1023. Camelcase Matching

Medium 808 285 Add to List Share

Given an array of strings `queries` and a string `pattern`, return a boolean array `answer` where `answer[i]` is `true` if `queries[i]` matches `pattern`, and `false` otherwise.

A query word `queries[i]` matches `pattern` if you can insert lowercase English letters pattern so that it equals the query. You may insert each character at any position and you may not insert any characters.

Example 1:

```
Input: queries =
["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"], pattern
= "FB"
Output: [true,false,true,true,false]
Explanation: "FooBar" can be generated like this "F" + "oo" + "B" + "ar".
"FootBall" can be generated like this "F" + "oot" + "B" + "all".
"FrameBuffer" can be generated like this "F" + "rame" + "B" + "uffer".
```

Example 2:

```
Input: queries =
["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"], pattern
= "FoBa"
Output: [true,false,true,false,false]
Explanation: "FooBar" can be generated like this "Fo" + "o" + "Ba" + "r".
"FootBall" can be generated like this "Fo" + "ot" + "Ba" + "ll".
```

Example 3:

```
Input: queries =
["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"], pattern
= "FoBaT"
Output: [false,true,false,false,false]
Explanation: "FooBarTest" can be generated like this "Fo" + "o" + "Ba" +
"r" + "T" + "est".
```

```cpp
class Solution {
public:
    bool check(string s,string t){
        int i=0,j=0;
        while(j<t.size() && i<s.size()){
            if(s[i]==t[j]) i++,j++;
            else if(s[i]>='a' && s[i]<='z') i++;
            else return false;
        }
        while(i<s.size()){
            if(s[i]>='a' && s[i]<='z') i++;
                else return false;
        }
        return (j==t.size());
    }
    vector<bool> camelMatch(vector<string>& queries, string pat) {
        vector<bool> ans;
        for(auto i:queries){
            ans.push_back(check(i,pat));
        }
        return ans;
    }
};
```

809. Expressive Words

Medium 816 1848 Add to List Share

Sometimes people repeat letters to represent extra feeling. For example:

- `"hello" -> "heeellooo"`
- `"hi" -> "hiiii"`

In these strings like `"heeellooo"`, we have groups of adjacent letters that are all the same: `"h"`, `"eee"`, `"ll"`, `"ooo"`.

You are given a string `s` and an array of query strings `words`. A query word is **stretchy** if it can be made to be equal to `s` by any number of applications of the following extension operation: choose a group consisting of characters `c`, and add some number of characters `c` to the group so that the size of the group is **three or more**.

- For example, starting with `"hello"`, we could do an extension on the group `"o"` to get `"hellooo"`, but we cannot get `"helloo"` since the group `"oo"` has a size less than three. Also, we could do another extension like `"ll" -> "llll"` to get `"helllllooo"`. If `s = "helllllooo"`, then the query word `"hello"` would be **stretchy** because of these two extension operations: `query = "hello" -> "hellooo" -> "helllllooo" = s`.

Return *the number of query strings that are **stretchy***.

Example 1:

```
Input: s = "heeellooo", words = ["hello", "hi", "helo"]
Output: 1
Explanation:
We can extend "e" and "o" in the word "hello" to get "heeellooo".
We can't extend "helo" to get "heeellooo" because the group "ll" is
not size 3 or more.
```

Example 2:

```
Input: s = "zzzzzyyyyy", words = ["zzyy","zy","zyy"]
Output: 3
```

```
var expressiveWords = function(S, words) {
  function count(str, i) {
    let start = i;
    while (i + 1 < str.length) {
      if (str[i] !== str[i + 1]) break;
      i++;
    }
    return i - start + 1;
  }

  function isStretchy(w) {
    let i = 0, j = 0;
    while (i < w.length && j < S.length) {
      if (w[i] !== S[j]) return false;
      let countA = count(w, i);
      let countB = count(S, j);
      if (countA > countB || (countA < countB && countB < 3)) {
        return false;
      }
      i += countA;
      j += countB;
    }
    return i == w.length && j == S.length;
  }

  let ans = 0;
  words.forEach(w => {
    if (isStretchy(w)) ans++;
  });
  return ans;
};
```

```
var expressiveWords = function(S, words) {
    const isExpressive = (word) => {
        let wI = 0;
        let sI = 0;

        while (wI < word.length || sI < S.length) {
            let countW = 1;
            let countS = 1;

            if (word[wI] !== S[sI]) return false;

            while (word[wI] === word[wI++ + 1]) countW++;
            while (S[sI] === S[sI++ + 1]) countS++;

            if (countS < countW || (countS !== countW && countS < 3)) return false;
        }


        return true;
    }

    return words.filter(isExpressive).length;
};
```


680. Valid Palindrome II

Easy 7507 381 Add to List Share

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
```


Two Pointer Approach

i is at 0th index;
j is at last index

while checking for i and j pointers if they are equal than just move the pointers.
if they aren't equal there could be two cases i != j
if we delete ith index character check for the i+1 to j are they a palindrome? if yes than the whole string is palindrome.
case 2: skip jth index character and see if i to j-1 are a palindrome or not

```
Time complexity - O(N)
Space complexity - O(1)
```

```cpp
class Solution {
public:
    bool ispalindrome(string s, int i, int j){
        while(i < j){
            if(s.at(i) == s.at(j)){
                i++;
                j--;
            }else return false;
        }
        return true;
    }
    bool validPalindrome(string s) {
        int i  = 0;
        int j = s.size()-1;
        while(i < j){
            if(s.at(i) == s.at(j)){
                i++;
                j--;
            }else{
                return ispalindrome(s, i+1, j) || ispalindrome(s, i, j-1);
            }
        }
        return true;
    }
};
```


43. Multiply Strings

Medium 6442 2958 Add to List Share

Given two non-negative integers `num1` and `num2` represented as strings, return the product of `num1` and `num2`, also represented as a string.

Note: You must not use any built-in BigInteger library or convert the inputs to integer directly.

Example 1:

```
Input: num1 = "2", num2 = "3"
Output: "6"
```

Example 2:

```
Input: num1 = "123", num2 = "456"
Output: "56088"
```


Constraints:

- `1 <= num1.length, num2.length <= 200`
- `num1` and `num2` consist of digits only.
- Both `num1` and `num2` do not contain any leading zero, except the number `0` itself.

```cpp
class Solution {
public:
    string multiply(string num1, string num2) {
        if (num1 == "0" || num2 == "0") return "0";
      string res = "";
      if(num1[0]=='-' && num2[0]=='-')
      {
          num1=num1.substr(1);
          num2=num2.substr(1);
      }
      else if(num1[0]=='-' && num2[0]!='-')
      {
          num1=num1.substr(1);
          res.push_back('-');
      }
      else if(num1[0]!='-' && num2[0]=='-')
      {
          num2=num2.substr(1);
          res.push_back('-');
      }
       vector<int> num(num1.size() + num2.size(), 0);
       for (int i = num1.size() - 1; i >= 0; --i) {
           for (int j = num2.size() - 1; j >= 0; --j) {
               num[i + j + 1] += (num1[i] - '0') * (num2[j] - '0');
               num[i + j] += num[i + j + 1] / 10;
               num[i + j + 1] %= 10;
           }
       }
       int i = 0;
       while (i < num.size() && num[i] == 0) ++i;
       while (i < num.size()) res.push_back(num[i++] + '0');
       return res;
    }
};
```



```cpp
class Solution {
public:
    string multiply(string A, string B) {
        int n = A.length(),m = B.length();
        string res(n+m,'0');

        for(int i=n-1;i>=0;i--){
            for(int j=m-1;j>=0;j--){
                int num = (A[i] - '0') * (B[j] - '0') + res[i+j+1] - '0';
                res[i+j+1] = num%10 + '0';
                res[i+j] += num/10;
            }
        }
        // cout<<res;
        for(int i=0;i<res.length();i++) if(res[i] != '0') return res.substr(i);
        return "0";
    }
};
```

777. Swap Adjacent in LR String

Medium 1149 897 Add to List Share

In a string composed of `'L'`, `'R'`, and `'X'` characters, like `"RXXLRXRXL"`, a move consists of either replacing one occurrence of `"XL"` with `"LX"`, or replacing one occurrence of `"RX"` with `"XR"`. Given the starting string `start` and the ending string `end`, return `True` if and only if there exists a sequence of moves to transform one string to the other.

Example 1:

```
Input: start = "RXXLRXRXL", end = "XRLXXRRLX"
Output: true
Explanation: We can transform start to end following these steps:
RXXLRXRXL ->
XRXLRXRXL ->
XRLXRXRXL ->
XRLXXRRXL ->
XRLXXRRLX
```

Example 2:

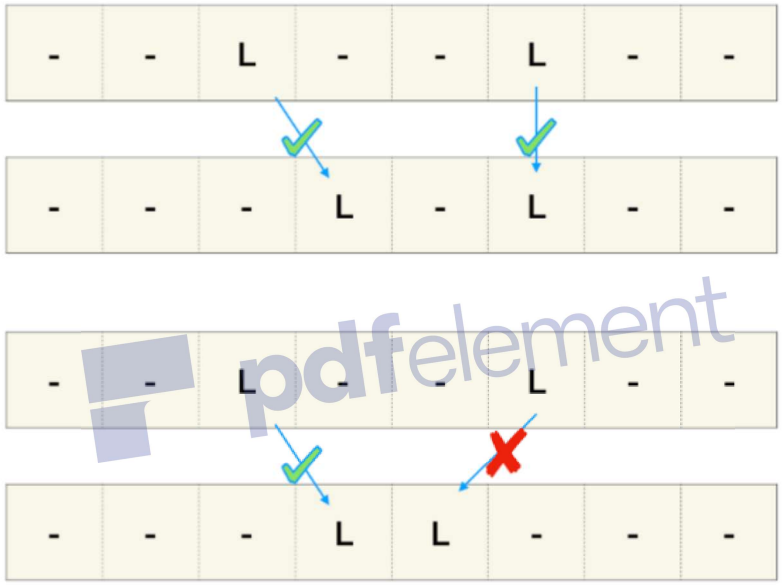
```
Input: start = "X", end = "L"
Output: false
```


Key observations:

There are three kinds of characters, 'L', 'R', 'X'.
Replacing XL with LX = move L to the left by one
Replacing RX with XR = move R to the right by one
If we remove all the X in both strings, the resulting strings should be the same.

Additional observations:

Since a move always involves X, an L or R cannot move through another L or R.
Since an L can only move to the right, for each occurrence of L in the start string, its position should be to the same or to the left of its corresponding L in the end string.



And vice versa for the R characters.

Implementation

We first compare two strings with X removed. This checks relative position between Ls and Rs are correct.

Then we find the indices for each occurence of L and check the condition in the above figure. Then we do the same for R.

Here I translate 'X' to '.' for understanding more easily.

```
start = "R..LR.R.L"
end   = ".RL..RRL."
```

From the above example, we can find some rules for 'L' and 'R':

1. "*XL*" to "*LX*": L only go left
2. "*RX*" to "*XR*": R only go right
3. L and R cannot change their order

For example, if we have "XXXXXXXL" and finally we may get the result "LXXXXXXX".

We only need to check the "L" and "R" position in two string.
Using two pointer `i` and `j` to keep comparing:

1. skip 'X', make sure `start[i]` and `end[j]` is the same
2. 'R' position in `start` is smaller than `end`
3. 'L' position in `start` is larger than `end`

```cpp
class Solution {
public:
    bool canTransform(string start, string end) {

        int i=0, j=0;
        while(i<start.size() && j<end.size()) {

            while(start[i]=='X') i++;
            while(end[j]=='X') j++;

            if(start[i]!=end[j]) return false;
            if(start[i]=='R' && i>j) return false;
            if(start[i]=='L' && i<j) return false;
            i++; j++;
        }
        while(i<start.size() && start[i]=='X') i++;
        while(j<end.size() && end[j]=='X') j++;
        return i==j;
    }
};
```

211. Design Add and Search Words Data Structure

Medium 7071 405 Add to List Share

Design a data structure that supports adding new words and finding if a string matches any previously added string.

Implement the `WordDictionary` class:

- `WordDictionary()` Initializes the object.
- `void addWord(word)` Adds `word` to the data structure, it can be matched later.
- `bool search(word)` Returns `true` if there is any string in the data structure that matches `word` or `false` otherwise. `word` may contain dots `'.'` where dots can be matched with any letter.

Example:

```
Input
["WordDictionary","addWord","addWord","addWord","search","search","search"
[[],["bad"],["dad"],["mad"],["pad"],["bad"],[".ad"],["b.."]]
Output
[null,null,null,null,false,true,true,true]

Explanation
WordDictionary wordDictionary = new WordDictionary();
wordDictionary.addWord("bad");
wordDictionary.addWord("dad");
wordDictionary.addWord("mad");
wordDictionary.search("pad"); // return False
wordDictionary.search("bad"); // return True
wordDictionary.search(".ad"); // return True
wordDictionary.search("b.."); // return True
```

```cpp
class WordDictionary {
    unordered_map<int, vector<string>> words;

    bool isEqual(string &word1, string &word2) {
        for (int i = 0; i < word1.size(); i++) {
            if (word2[i] == '.')
                continue;
            if (word1[i] != word2[i])
                return false;
        }
        return true;
    }

public:
    WordDictionary() {}

    void addWord(string word) {
        words[word.size()].push_back(word);
    }

    bool search(string word) {
        for (auto &&s : words[word.size()])
            if (isEqual(s, word))
                return true;
        return false;
    }
};
/**
 * Your WordDictionary object will be instantiated and called as such:
 * WordDictionary* obj = new WordDictionary();
 * obj->addWord(word);
 * bool param_2 = obj->search(word);
 */
```


1163. Last Substring in Lexicographical Order

Hard 545 438 Add to List Share

Given a string `s`, return *the last substring of* `s` *in lexicographical order.*

Example 1:

```
Input: s = "abab"
Output: "bab"
Explanation: The substrings are ["a", "ab", "aba", "abab", "b", "ba",
"bab"]. The lexicographically maximum substring is "bab".
```

Example 2:

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


Constraints:

- `1 <= s.length <= 4 * 10^5`
- `s` contains only lowercase English letters.

```
class Solution {
public:
// We use "j" to find a better starting index. If any is found, we use it to update "i"

// 1."i" is the starting index of the first substring
// 2."j" is the staring index of the second substring
// 3."k" is related to substring.length() (eg. "k" is substring.length()-1)

// Case 1 (s[i+k]==s[j+k]):
// -> If s.substr(i,k+1)==s.substr(j,k+1), we keep increasing k.
// Case 2 (s[i+k]<s[j+k]):
// -> If the second substring is larger, we update i with max(i+k+1,j).
//Since we can skip already matched things (The final answer is s.substr(i))
// Case 3 (s[i+k]>s[j+k]):
// -> If the second substring is smaller, we just change the starting index of the second string to j+k+1.
//Because s[j]~s[j+k] must be less than s[i], otherwise "i" will be updated by "j". So the next possible candidate is "j+k+1".

    string lastSubstring(string s) {
        int n=s.length(),i=0,j=1,k=0;
        while(j+k<n)
        {
            if(s[i+k]==s[j+k]) k++;
            else if(s[i+k]<s[j+k]) i=max(i+k+1,j),j=i+1,k=0;
            else j+=k+1,k=0;
        }
        return s.substr(i);
    }
}
```

```
class Solution {
    public:
    string lastSubstring(string s) {
        int maxIndex = s.length() - 1;

        for (int currIndex = s.length() - 1; currIndex >= 0; currIndex--) {
            if (s[currIndex] > s[maxIndex])
                maxIndex = currIndex;

            else if (s[currIndex] == s[maxIndex]) {
                int i = currIndex + 1;
                int j = maxIndex + 1;

                while (i < maxIndex && j < s.length() && s[i] == s[j]) {
                    i++;
                    j++;
                }

                if (i == maxIndex || j == s.length() || s[i] > s[j])
                    maxIndex = currIndex;
            }
        }

        return s.substr(maxIndex);
    }
};
```

```
class Solution {
public:
    string lastSubstring(string s) {
        //find the first index with the lexicographically biggest value
        int biggest = 0;
        for (int i = 1; i < s.size(); i++)  {
            if (s[i] > s[biggest])
                biggest = i;
        }

    /*  from that index onwards, find indices with the same letter
        if found, run a comparison of the two strings, indicated by:
        j represents the current maximum string (from [biggest, s.size()))
        j = [biggest, start_i)
        start_i is the index we just encountered
        i = [start_i, s.size())
        i represents the string that could be lexicographically larger  */
        for (int i = biggest + 1; i < s.size(); i++) {
            if (s[i] == s[biggest]) {
                int j = biggest;
                int start_i = i;
                //we keep incrementing i & j till they're in range and the chars at their place are equal
                while (i < s.size() && j < start_i && s[i] == s[j]) {
                    i++, j++;
                }
                //if i == s.size() ==> both strings are equal
                //if not, and j < start_i, that means we encountered a point of non-equality of chars
                //now if at this point, we encountered a char that is bigger, we have a new max_string
                //otherwise j==start_i and the previous string is already the biggest one
                if (i < s.size() && j < start_i) {
                    if (s[i] > s[j])
                        biggest = start_i;
                }
                i--;
            }
        }
        return s.substr(biggest);
    }
};
```