DSA question solving leetcode

array

16. 3Sum Closest

Medium

Topics

Companies

Given an integer array `nums` of length `n` and an integer `target`, find three integers in `nums` such that the sum is closest to `target`.

Return *the sum of the three integers*.

You may assume that each input would have exactly one solution.

Example 1:

```
Input: nums = [-1,2,1,-4], target = 1
Output: 2
Explanation: The sum that is closest to the target is 2. (-1
+ 2 + 1 = 2).
```

Example 2:

```
Input: nums = [0,0,0], target = 1
Output: 0
Explanation: The sum that is closest to the target is 0. (0 +
0 + 0 = 0).
```

Constraints:

- `3 <= nums.length <= 500`

- `1000 <= nums[i] <= 1000`
- `104 <= target <= 104`

code :

topics included

- two pointer
- sorting

```
class Solution {
    public:
        int threeSumClosest(vector<int>& nums, int target) {
            sort(nums.begin(), nums.end());
            int res = 0;
            int gap = INT_MAX;

            for(int i=0;i<nums.size() - 2 ;i++){
                int li = i+ 1 ;
                int ri = nums.size() -1 ;

                while(li < ri){
                    int sum = nums[i] + nums[li] + nums[ri];
                    int newgap = abs(target - sum);
                    if (sum < target) {
                        if (newgap < gap ){
                            gap = min(gap , newgap);
                            res = sum;
                        }
                        li++ ;
                    }
                    else if( sum > target ){
                        if(newgap < gap){
                            gap = min(gap , newgap);
```

```
                            res = sum;
                        }
                        ri--;
                    }
                    else{
                        return sum;
                    }
                }
            }
            return res;
        }
};
```

713. Subarray Product Less Than K

Medium

Topics

Companies

Hint

Given an array of integers `nums` and an integer `k`, return *the number of contiguous subarrays where the product of all the elements in the subarray is strictly less than* `k`.

Example 1:

```
Input: nums = [10,5,2,6], k = 100
Output: 8
Explanation: The 8 subarrays that have product less than 100
are:
[10], [5], [2], [6], [10, 5], [5, 2], [2, 6], [5, 2, 6]
Note that [10, 5, 2] is not included as the product of 100 is
not strictly less than k.
```

Example 2:

```
Input: nums = [1,2,3], k = 0
Output: 0
```

Constraints:

- `1 <= nums.length <= 3 * 104`
- `1 <= nums[i] <= 1000`
- `0 <= k <= 106`

Code:

```
class Solution {
    public:
        int numSubarrayProductLessThanK(vector<int>& nums, int k

            int left = 0, right = 0 ;
            long long currentProd = 1 ;

            int cnt = 0;
            while(right < nums.size() ){
                currentProd *= nums[right];

                while(currentProd > k -1  and left <= right){
                    currentProd /= nums[left];
                    left ++ ;
                }

                if (currentProd <= k - 1) {
                    cnt += right - left + 1 ;
                }

                right++ ;
```

```
            }
            return cnt;
        }
};
```

This code is a solution to the "Subarray Product Less Than K" problem using a sliding window approach. Here's a detailed explanation of how it works:

Variables and Initialization:

1. `left` and `right`: These are two pointers used to define the boundaries of the sliding window.
2. `currentProd`: This variable holds the product of all elements within the current window (`nums[left]` to `nums[right]`). It's initialized to 1.
3. `cnt`: This variable counts the number of valid subarrays where the product of elements is less than `k`.

The Main Logic:

The main part of the code uses a `while` loop to slide the `right` pointer through the array. As `right` moves, the product of the current window is updated. Here's a step-by-step breakdown:

1. **Expand the Window**:
   - `currentProd *= nums[right];`
   - Multiply `currentProd` by the current element (`nums[right]`).
2. **Shrink the Window**:
   - The inner `while` loop checks if `currentProd` is greater than or equal to `k`.
   - If `currentProd` exceeds or equals `k`, you need to shrink the window from the left by moving the `left` pointer to the right.
   - You divide `currentProd` by `nums[left]` to remove the effect of `nums[left]` from the product and then increment `left`.
3. **Count Valid Subarrays**:

- If `currentProd` is less than `k`, the number of valid subarrays ending at `right` is `(right - left + 1)`. This is because every subarray that starts from any index between `left` and `right` (inclusive) and ends at `right` is valid.
- Add this number to `cnt`.

4. **Move the `right` Pointer**:
   - Increment `right` to expand the window and continue the process.

Return Value:

- After the loop finishes, the function returns `cnt`, which contains the total number of subarrays whose product is less than `k`.

Key Points:

- **Sliding Window**: The solution efficiently finds subarrays by adjusting the window size dynamically, rather than checking all possible subarrays explicitly.
- **Time Complexity**: The time complexity is **O(n)** because each element is processed at most twice (once when expanding the window and once when contracting it).
- **Edge Cases**: The condition `currentProd > k - 1` ensures that the product is strictly less than `k`.

This code is optimal for solving the problem within the constraints provided.

828. Count Unique Characters of All Substrings of a Given String

Hard

Topics

Companies

Let's define a function `countUniqueChars(s)` that returns the number of unique characters in `s`.

- For example, calling `countUniqueChars(s)` if `s = "LEETCODE"` then `"L"`, `"T"`, `"C"`, `"O"`, `"D"` are the unique characters since they appear only once in `s`, therefore `countUniqueChars(s) = 5`.

Given a string `s`, return the sum of `countUniqueChars(t)` where `t` is a substring of `s`. The test cases are generated such that the answer fits in a 32-bit integer.

Notice that some substrings can be repeated so in this case you have to count the repeated ones too.

Example 1:

```
Input: s = "ABC"
Output: 10
Explanation:All possible substrings are: "A","B","C","AB","B
C" and "ABC".
Every substring is composed with only unique letters.
Sum of lengths of all substring is 1 + 1 + 1 + 2 + 2 + 3 = 10
```

Example 2:

```
Input: s = "ABA"
Output: 8
Explanation:The same as example 1, exceptcountUniqueChars("AB
A") = 1.
```

Example 3:

```
Input: s = "LEETCODE"
Output: 92
```

Constraints:

- `1 <= s.length <= 105`
- `s` consists of uppercase English letters only.
- topics : harsh table , strings , dynampic programing

code :

630. Course Schedule III

Hard

Topics

Companies

Hint

There are `n` different online courses numbered from `1` to `n`. You are given an array `courses` where `courses[i] = [durationi, lastDayi]` indicate that the `ith` course should be taken **continuously** for `durationi` days and must be finished before or on `lastDayi`.

You will start on the `1st` day and you cannot take two or more courses simultaneously.

Return *the maximum number of courses that you can take*.

Example 1:

```
Input: courses = [[100,200],[200,1300],[1000,1250],[2000,320
0]]
Output: 3
Explanation:
There are totally 4 courses, but you can take 3 courses at mo
st:
First, take the 1st course, it costs 100 days so you will fin
ish it on the 100th day, and ready to take the next course on
the 101st day.
Second, take the 3rd course, it costs 1000 days so you will f
inish it on the 1100th day, and ready to take the next course
on the 1101st day.
Third, take the 2nd course, it costs 200 days so you will fin
ish it on the 1300th day.
```

```
The 4th course cannot be taken now, since you will finish it
on the 3300th day, which exceeds the closed date.
```

Example 2:

```
Input: courses = [[1,2]]
Output: 1
```

Example 3:

```
Input: courses = [[3,2],[4,3]]
Output: 0
```

Constraints:

- `1 <= courses.length <= 104`
- `1 <= durationi, lastDayi <= 104`

topics : array , Greedy, Sorting , dynamic programming .

252. Meeting Rooms

Given an array of meeting time intervals consisting of start and end times `[[s1,e1],[s2,e2],...]` (si < ei), determine if a person could attend all meetings.

Example 1:

```
Input:[[0,30],[5,10],[15,20]]Output: false
```

Example 2:

```
Input: [[7,10],[2,4]]
Output: true
```

NOTE: input types have been changed on April 15, 2019. Please reset to default code definition to get new method signature.

Difficulty:Easy

Lock:Prime

Company:Amazon Bloomberg Facebook Google Microsoft Twitter

88. Merge Sorted Array

Easy

Topics

Companies

Hint

You are given two integer arrays `nums1` and `nums2`, sorted in **non-decreasing order**, and two integers `m` and `n`, representing the number of elements in `nums1` and `nums2` respectively.

Merge `nums1` and `nums2` into a single array sorted in **non-decreasing order**.

The final sorted array should not be returned by the function, but instead be *stored inside the array* `nums1`. To accommodate this, `nums1` has a length of `m + n`, where the first `m` elements denote the elements that should be merged, and the last `n` elements are set to `0` and should be ignored. `nums2` has a length of `n`.

Example 1:

```
Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
Output: [1,2,2,3,5,6]
Explanation: The arrays we are merging are [1,2,3] and [2,5,
6].
The result of the merge is [1,2,2,3,5,6] with the underlined
elements coming from nums1.
```

Example 2:

```
Input: nums1 = [1], m = 1, nums2 = [], n = 0
Output: [1]
Explanation: The arrays we are merging are [1] and [].
The result of the merge is [1].
```

Example 3:

```
Input: nums1 = [0], m = 0, nums2 = [1], n = 1
Output: [1]
Explanation: The arrays we are merging are [] and [1].
The result of the merge is [1].
Note that because m = 0, there are no elements in nums1. The
0 is only there to ensure the merge result can fit in nums1.
```

Constraints:

- `nums1.length == m + n`
- `nums2.length == n`
- `0 <= m, n <= 200`
- `1 <= m + n <= 200`
- `109 <= nums1[i], nums2[j] <= 109`

Follow up: Can you come up with an algorithm that runs in `O(m + n)` time?

code :

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, in
        int last = m + n - 1;  // Index of the last element in n

        // Start from the end of both arrays and merge them into
        while (m > 0 && n > 0) {
            if (nums1[m - 1] > nums2[n - 1]) {
```

```
                nums1[last] = nums1[m - 1];
                m--;
            } else {
                nums1[last] = nums2[n - 1];
                n--;
            }
            last--;
        }

        // If there are still elements left in nums2, copy them
        while (n > 0) {
            nums1[last] = nums2[n - 1];
            n--;
            last--;
        }
    }
};
```

27. Remove Element

Solved

Easy

Topics

Companies

Hint

Given an integer array `nums` and an integer `val`, remove all occurrences of `val` in `nums` **in-place**. The order of the elements may be changed. Then return *the number of elements in* `nums` *which are not equal to* `val`.

Consider the number of elements in `nums` which are not equal to `val` be `k`, to get accepted, you need to do the following things:

- Change the array `nums` such that the first `k` elements of `nums` contain the elements which are not equal to `val`. The remaining elements of `nums` are not

important as well as the size of `nums`.

- Return `k`.

Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int val = ...; // Value to remove
int[] expectedNums = [...]; // The expected answer with corre
ct length.
                            // It is sorted with no values eq
ualing val.

int k = removeElement(nums, val); // Calls your implementatio
n

assert k == expectedNums.length;
sort(nums, 0, k); // Sort the first k elements of nums
for (int i = 0; i < actualLength; i++) {
    assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be **accepted**.

Example 1:

```
Input: nums = [3,2,2,3], val = 3
Output: 2, nums = [2,2,_,_]
Explanation: Your function should return k = 2, with the firs
t two elements of nums being 2.
It does not matter what you leave beyond the returned k (henc
e they are underscores).
```

Example 2:

```
Input: nums = [0,1,2,2,3,0,4,2], val = 2
Output: 5, nums = [0,1,4,0,3,_,_,_]
Explanation: Your function should return k = 5, with the firs
t five elements of nums containing 0, 0, 1, 3, and 4.
Note that the five elements can be returned in any order.
It does not matter what you leave beyond the returned k (henc
e they are underscores).
```

Constraints:

- `0 <= nums.length <= 100`
- `0 <= nums[i] <= 50`
- `0 <= val <= 100`

code :

```
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        int left = 0, right = 0;
        int n = nums.size();

    while (left < n) {
        if (nums[left] != val) {
            swap(nums[left], nums[right]);
            right++;
        }
        left++;
    }

    return right;
    }
};
```

https://leetcode.com/tag/sliding-window/

26. Remove Duplicates from Sorted Array

Solved

Easy

Topics

Companies

Hint

Given an integer array `nums` sorted in **non-decreasing order**, remove the duplicates **in-place** such that each unique element appears only **once**.
The **relative order** of the elements should be kept the **same**. Then return *the number of unique elements in* `nums`.

Consider the number of unique elements of `nums` to be `k`, to get accepted, you need to do the following things:

- Change the array `nums` such that the first `k` elements of `nums` contain the unique elements in the order they were present in `nums` initially. The remaining elements of `nums` are not important as well as the size of `nums`.
- Return `k`.

Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int[] expectedNums = [...]; // The expected answer with corre
ct length

int k = removeDuplicates(nums); // Calls your implementation
```

```
assert k == expectedNums.length;
for (int i = 0; i < k; i++) {
    assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be **accepted**.

Example 1:

```
Input: nums = [1,1,2]
Output: 2, nums = [1,2,_]
Explanation: Your function should return k = 2, with the firs
t two elements of nums being 1 and 2 respectively.
It does not matter what you leave beyond the returned k (henc
e they are underscores).
```

Example 2:

```
Input: nums = [0,0,1,1,1,2,2,3,3,4]
Output: 5, nums = [0,1,2,3,4,_,_,_,_,_]
Explanation: Your function should return k = 5, with the firs
t five elements of nums being 0, 1, 2, 3, and 4 respectively.
It does not matter what you leave beyond the returned k (henc
e they are underscores).
```

Constraints:

- `1 <= nums.length <= 3 * 104`
- `100 <= nums[i] <= 100`
- `nums` is sorted in **non-decreasing** order.

code :

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if (nums.size() == 0) return 0;

        int left = 0;

        for (int right = 1; right < nums.size(); right++) {
            if (nums[right] != nums[left]) {
                left++;
                nums[left] = nums[right];
            }
        }

        return left + 1;
    }
};
```

Explanation:

1. **Initial Check**:
   - If the array is empty (`nums.size() == 0`), return 0.
2. **Two Pointers**:
   - The `left` pointer marks the position to overwrite, and the `right` pointer iterates through the array.
   - When `nums[right]` is different from `nums[left]`, increment `left` and update `nums[left]` with `nums[right]`.
3. **Return Value**:
   - After the loop, `left + 1` gives the length of the array without duplicates.

How It Works:

- The array is modified in place, and all unique elements are moved to the front. The function then returns the length of the unique elements, and elements beyond this length are not considered in the final result.

Example:

For the input `nums = [1, 1, 2]`, the output will be `2`, and the array will be modified to `[1, 2, ...]`.

Time Complexity:

- The time complexity is **O(n)**, where `n` is the length of the input array. The solution uses constant extra space.

---

13. Roman to Integer

Easy

Topics

Companies

Hint

Roman numerals are represented by seven different symbols: `I`, `V`, `X`, `L`, `C`, `D` and `M`.

```
SymbolValue
I             1
V             5
X             10
L             50
C             100
D             500
M             1000
```

For example, `2` is written as `II` in Roman numeral, just two ones added together. `12` is written as `XII`, which is simply `X + II`. The number `27` is written

as `XXVII`, which is `XX + V + II`.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not `IIII`. Instead, the number four is written as `IV`. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as `IX`. There are six instances where subtraction is used:

- `I` can be placed before `V` (5) and `X` (10) to make 4 and 9.
- `X` can be placed before `L` (50) and `C` (100) to make 40 and 90.
- `C` can be placed before `D` (500) and `M` (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

Example 1:

```
Input: s = "III"
Output: 3
Explanation: III = 3.
```

Example 2:

```
Input: s = "LVIII"
Output: 58
Explanation: L = 50, V= 5, III = 3.
```

Example 3:

```
Input: s = "MCMXCIV"
Output: 1994
Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.
```

Constraints:

- `1 <= s.length <= 15`
- `s` contains only the characters `('I', 'V', 'X', 'L', 'C', 'D', 'M')`.
- It is **guaranteed** that `s` is a valid roman numeral in the range `[1, 3999]`.

code :

```
class Solution {
public:
    int romanToInt(string s) {
        unordered_map<char, int> roman = {
            {'I', 1}, {'V', 5}, {'X', 10},
            {'L', 50}, {'C', 100}, {'D', 500}, {'M', 1000}
        };

        int total = 0;
        int n = s.size();

        for (int i = 0; i < n; i++) {
            // If this numeral is less than the next one, subtra
            if (i < n - 1 && roman[s[i]] < roman[s[i + 1]]) {
                total -= roman[s[i]];
            } else {
                total += roman[s[i]];
            }
        }

        return total;
    }
};
```

Explanation:

1. **Mapping Roman Numerals**:
    - We use an `unordered_map` to store the integer values for each Roman numeral character.
2. **Iterate Over the String**:
    - We iterate through the string. For each character:

    - If it’s followed by a numeral of a higher value (indicating a subtractive combination), subtract its value from the total.
    - Otherwise, add its value to the total.
3. **Return the Total**:
    - After processing all characters in the string, the total will be the integer equivalent of the Roman numeral.

Example Walkthrough:

For `s = "MCMXCIV"`:

- Start with `total = 0`.
- M = 1000 → `total = 1000`
- C = 100 and M = 1000 (C < M) → `total = 1000 - 100 = 900`
- M = 1000 → `total = 900 + 1000 = 1900`
- X = 10 and C = 100 (X < C) → `total = 1900 - 10 = 1890`
- C = 100 → `total = 1890 + 100 = 1990`
- I = 1 and V = 5 (I < V) → `total = 1990 - 1 = 1989`
- V = 5 → `total = 1989 + 5 = 1994`

Edge Cases:

- Strings like "III" or "LVIII" that don't involve subtraction.
- A string with a single Roman numeral, e.g., "I".

Time Complexity:

- The time complexity is **O(n)**, where `n` is the length of the string `s`, because we are iterating over the string once.

This approach correctly handles all cases and efficiently converts a Roman numeral to its integer equivalent.

4o

58. Length of Last Word

Easy

Topics

Companies

Given a string `s` consisting of words and spaces, return *the length of the **last** word in the string.*

A **word** is a maximal

substring

consisting of non-space characters only.

Example 1:

```
Input: s = "Hello World"
Output: 5
Explanation: The last word is "World" with length 5.
```

Example 2:

```
Input: s = "   fly me   to   the moon  "
Output: 4
Explanation: The last word is "moon" with length 4.
```

Example 3:

```
Input: s = "luffy is still joyboy"
Output: 6
Explanation: The last word is "joyboy" with length 6.
```

Constraints:

- `1 <= s.length <= 104`
- `s` consists of only English letters and spaces `' '`.
- There will be at least one word in `s`.

code :

```
class Solution {
public:
    int lengthOfLastWord(string s) {
       int left = s.size() -1 ;
       int cnt = 0;
       while(left >= 0 and s[left] == ' '){
        left --;
       }
       while(left>=0 and s[left] != ' '){
        cnt ++ ;
        left--;
       }
       return cnt;

    }
};
```

---

14. Longest Common Prefix

Easy

Topics

Companies

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 string
s.
```

Constraints:

- `1 <= strs.length <= 200`
- `0 <= strs[i].length <= 200`
- `strs[i]` consists of only lowercase English letters.

code:

```
string longestCommonPrefix(vector<string>& strs) {
    if (strs.empty()) {
        return "";
    }

    string prefix = strs[0];

    for (int i = 1; i < strs.size(); ++i) {
        int j = 0;
        while (j < prefix.length() && j < strs[i].length() && pr
            ++j;
        }
        prefix = prefix.substr(0, j);  // Update the prefix to t
        if (prefix.empty()) {
            break;  // Early exit if there is no common prefix
        }
    }

    return prefix;
}
```

Explanation:

1. **Initialization**:
   - We start by assuming the first string in the list (`strs[0]`) is the common prefix.
2. **Iterate Through the Array**:
   - For each subsequent string in the array, we compare it with the current `prefix`.
   - The comparison is done character by character. We update the `prefix` to be the common part between the `prefix` and the current string.
3. **Early Termination**:
   - If at any point the common prefix becomes empty (i.e., there's no common prefix between the strings), we break out of the loop early.
4. **Return**:
   - After processing all strings, the `prefix` will contain the longest common prefix.

Example Execution:

- For the input `["flower", "flow", "flight"]`, the algorithm starts with `prefix = "flower"`. After comparing with `"flow"` and `"flight"`, the `prefix` is reduced to `"fl"`.
- For the input `["dog", "racecar", "car"]`, the algorithm quickly determines that there is no common prefix and returns an empty string.

This approach efficiently finds the longest common prefix in O(N * M) time, where N is the number of strings and M is the length of the shortest string.

4o

28. Find the Index of the First Occurrence in a String

Easy

Topics

Companies

Given two strings `needle` and `haystack`, return the index of the first occurrence of `needle` in `haystack`, or `-1` if `needle` is not part of `haystack`.

Example 1:

```
Input: haystack = "sadbutsad", needle = "sad"
Output: 0
Explanation: "sad" occurs at index 0 and 6.
The first occurrence is at index 0, so we return 0.
```

Example 2:

```
Input: haystack = "leetcode", needle = "leeto"
Output: -1
Explanation: "leeto" did not occur in "leetcode", so we retur
n -1.
```

Constraints:

- `1 <= haystack.length, needle.length <= 104`
- `haystack` and `needle` consist of only lowercase English characters.

code :

```
class Solution {
    public:
        int strStr(string haystack, string needle) {
            int n = haystack.size();
```

```
            int m = needle.size();

            if (m == 0) {
                return 0;  // edge case: empty needle
            }

            for (int i = 0; i <= n - m; ++i) {
                if (haystack.substr(i, m) == needle) {
                    return i;
                }
            }

            return -1;  // needle not found

        }
};
```

1. **Loop through `haystack`**:
    - The loop runs from `i = 0` to `i = n - m`, where `n` is the length of `haystack` and `m` is the length of `needle`. This ensures we don't go out of bounds when checking the substring.
2. **Substring Check**:
    - We use `haystack.substr(i, m)` to extract the substring of length `m` starting from index `i` and compare it with `needle`.
3. **Return Index**:
    - If a match is found, we return the index `i`.
    - If no match is found, we return `-1`.

Example Execution:

- **Example 1**: For `haystack = "sadbutsad"` and `needle = "sad"`, the first occurrence is at index `0`, so the output is `0`.
- **Example 2**: For `haystack = "leetcode"` and `needle = "leeto"`, the `needle` is not found, so the output is `1`.

This approach has a time complexity of O(N * M), where N is the length of `haystack` and M is the length of `needle`. While this is efficient for small inputs, more advanced algorithms like the KMP (Knuth-Morris-Pratt) algorithm can be used for faster string matching, especially in larger inputs.