Leetstreak

By Sudarshan T S

**LeetCode 2239:** **Find Closest Number to Zero** 28/10/2024

Given an integer array nums of size n, return *the number with the value closest to*0*in*nums. If there are multiple answers, return *the number with the largest value*.

**Solution:**

**Approach 1:**It updates the closest number by comparing the absolute values and favoring positive numbers in case of ties.

**Time Complexity:**

* **O(n)**, where *n* is the length of the list, as it requires a single pass through all elements.

**Space Complexity:**

* **O(1)**, since it uses only a constant amount of extra space regardless of the input size.

**Code:**

if not nums: # O(1)  
 return None  
  
# Initialize closest number as the first element  
closest = nums[0]  
  
# If the abs value of the current number is smaller, or if  
# the same but positive  
for num in nums[1:]: # O(n), n = length of the list  
 if abs(num) < abs(closest) or (abs(num) == abs(closest) and num > closest):  
 closest = num  
  
return closest

**Approach 2:**

It uses min() with a lambda function as the key to find the closest number to zero. The lambda function sorts by absolute value and, in case of a tie, prefers positive numbers.

**Time Complexity:**

* **O(n),** where *n* is the length of the list, as it scans the list once.

**Space Complexity:**

* **O(1),** as it only uses a constant amount of extra space.

**Code:**

def findClosestNumber(nums):

if not nums: # Check for empty list

return None

return min(nums, key=lambda x: (abs(x), -x))

**LeetCode** [**1768: Merge Strings Alternately**](https://leetcode.com/problems/merge-strings-alternately/) 28/10/2024

You are given two strings word1 and word2. Merge the strings by adding letters in alternating order, starting with word1. If a string is longer than the other, append the additional letters onto the end of the merged string.

Return *the merged string.*

**Solution:**

**Approach 1:**

1. Convert the input strings word1 and word2 into lists a and b.
2. Initialize an empty list result to store the merged characters.
3. Determine the minimum length m of the two lists.
4. Loop through the first m characters, appending one character from each list to the result in alternating order.
5. If one of the lists has remaining characters (when len(a) != len(b)), append these remaining characters to the result.
6. Join the characters in the result list to form a single string and return it.

**Time Complexity:**

* **O(n + m),** where n and m are the lengths of word1 and word2. This complexity arises because:
  + We iterate through the shorter string (min(len(word1), len(word2))).
  + We append the remaining characters from the longer string.

**Space Complexity:**

* **O(n + m),** where n and m are the lengths of the two input strings. This is due to the creation of the result list which stores all the characters from both strings.

**Code:**

a = list(word1) # converting string into list

        b = list(word2) # converting string into list

        result = [] # an empty list for result

        m = min(len(a),len(b)) # number of iteration for appending one letter from each list

        for i in range(m):

            result.append(a[i])

            result.append(b[i])

        if len(a)>len(b):

            x = a[m :] # rest of letters in list a

            result.append(''.join(x))

        elif len(a)<len(b):

            x = b[m :] # rest of letters in list b

            result.append(''.join(x))

        return (''.join(result))

**Approach 2:**

1. **Loop through the Shorter String**: Append characters from both strings alternately to the result list until the end of the shorter string.
2. **Append Remaining Characters**: Add leftover characters from the longer string to the result list.
3. **Return the Combined Result**: Join the result list into a single string.

**Time Complexity:**

* **O(n + m)**, where n and m are the lengths of the two input strings.

**Space Complexity:**

* **O(n + m)**, for storing the result list which contains all characters from both strings.

**Code:**

def mergeAlternately(word1, word2):

result = []

# Iterate through the minimum length of both words

for i in range(min(len(word1), len(word2))):

result.append(word1[i])

result.append(word2[i])

# Add remaining characters from the longer string

result.append(word1[i + 1:] if len(word1) > len(word2) else word2[i + 1:])

return ''.join(result)

**LeetCode 13: Roman to integer** 28/10/2024

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

**Symbol** **Value**

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.

**Approach 1:**

* This approach uses a combination of a dictionary and a list to map and compare Roman numeral values, iterating through the string in reverse.

**Time Complexity:**

* **O(n)**, where *n* is the length of the Roman numeral string. The function processes each character in the string exactly once.

**Space Complexity:**

* **O(1)**, since the space used for the dictionary and list does not scale with the input size, and only a fixed amount of additional space is used for variables.

**Code:**

def roman\_to\_int(s):  
 # dictiory of roman value to int  
 roman = { "I":1,"V":5, "X":10,"L":50,"C":100,"D":500,"M":1000 }  
 j = len(s)-1 # to access from last index  
 result = 0  
 rom = ["I","V","X","L","C","D","M"] #to check with index values  
  
 result+=roman[s[j]] #adding last value directly  
 #iterate all the values from last till j is grater than 0  
 while j>0:  
 if rom.index(s[j])>rom.index(s[j-1]):  
 result -= roman[s[j-1]]  
 else:  
 result+= roman[s[j-1]]  
 j-=1  
 return result

**Approach 2:**

It iterates through the string in reverse, adding or subtracting values based on their comparison with the previous numeral to handle additive and subtractive cases.

**Time Complexity:**

* **O(n)**, where *n* is the length of the input string.

**Space Complexity:**

* **O(1)**, as it uses a constant amount of space for variables and the dictionary.

**Code:**

def roman\_to\_int(s):

roman = { "I": 1, "V": 5, "X": 10, "L": 50, "C": 100, "D": 500, "M": 1000 }

result = 0

prev\_value = 0 # Track the value of the previous numeral

for char in reversed(s):

current\_value = roman[char]

if current\_value < prev\_value:

result -= current\_value # Subtract if the current value is less than the previous

else:

result += current\_value # Otherwise, add the current value

prev\_value = current\_value # Update the previous value for the next iteration

return result

**LeetCode 392: Is Subsequence** 28/10/2024

Given two strings s and t, return true*if*s*is a****subsequence****of*t*, or*false*otherwise*.

A **subsequence** of a string is a new string that is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (i.e., "ace" is a subsequence of "abcde" while "aec" is not).

**Approach 1:**

The code uses two pointers to traverse both strings. It matches characters of `s` with `t` in order, incrementing both pointers when a match is found. After traversal, it checks if all characters in `s` were matched.

**Time Complexity:**

* **O(n + m)** where n is the length of `s` and m is the length of `t`.

**Space Complexity:**

* **O(1)**

**Code:**

def is\_subsequence(s, t):  
 i, j = 0, 0  
  
 # Iterate while there are characters left in both strings  
 while i < len(s) and j < len(t):  
 if s[i] == t[j]:  
 # If characters match, move both pointers forward  
 i += 1  
 # Move the pointer for string t forward in all cases  
 j += 1  
  
 # Check if all characters of s were matched  
 return i == len(s)

**Approach 2:**

The function uses a **two-pointer technique** to traverse both strings s and t. It iterates through t and attempts to match the characters of s in order:

* If the current character of t matches the current character of s, move to the next character in s.
* If all characters of s are successfully matched, return True.
* If the loop ends without matching all characters of s, return False.

**Time Complexity:**

* **O(T)**, where *T* is the length of t. The function iterates through the string t only once.

**Space Complexity:**

* **O(1)**, as it uses a few pointers (i and j) and constant space, independent of the input size.

**Code:**

def isSubsequence( s, t):

S = len(s)

T = len(t)

if s == '': return True

if S > T: return False

j = 0

for i in range(T):

if t[i] == s[j]:

if j == S-1:

return True

j += 1

return False

**LeetCode 121: Best Time to Buy and Sell Stock** 29/10/2024

You are given an array prices where prices[i] is the price of a given stock on the ith day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

**Approach 1:**

Iterate through the prices to find the minimum price so far and calculate the potential profit at each step. Update the maximum profit accordingly.

**Time Complexity:**

* **O(n),** where *n* is the number of prices.

**Space Complexity:**

* **O(1),** as only a few variables are used.

**Code:**

def bestbuy(prices):

# Initialize minimum price to infinity and maximum profit to 0

min\_price = float("inf")

max\_profit = 0

# Iterate through each price in the list

for price in prices:

# Update minimum price if the current price is lower

if price < min\_price:

min\_price = price

# Calculate the profit with the current price

profit = price - min\_price

# Update maximum profit if the current profit is higher

if profit > max\_profit:

max\_profit = profit

return max\_profit

**LeetCode** [**14: Longest Common Prefix**](https://leetcode.com/problems/longest-common-prefix/) 29/10/2024

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 "".

**Approach 1:**

1. **Handle Edge Cases**: Return an empty string if strs is empty.
2. **Find Shortest String**: Identify the shortest string in the list as the maximum possible prefix length.
3. **Character Comparison**: Compare each character in the shortest string with corresponding characters in all strings. Return the prefix when a mismatch is found.
4. **Return Result**: If no mismatches occur, return the shortest string.

**Time Complexity:**

* **O(n \* m)**, where *n* is the number of strings and *m* is the length of the shortest string.

**Space Complexity:**

* **O(1)**, using a constant amount of extra space

**Code:**

def longestCommonPrefix(strs):

if not strs:

return "" # Return empty string if the list is empty

# Find the shortest string in the list

check = min(strs, key=len)

for i in range(len(check)):

for word in strs:

if word[i] != check[i]:

return check[:i] # Return common prefix up to this point

return check # All characters matched, return the shortest string

**Approach 2:**

* Start with the first string as the prefix.
* For each string, trim the prefix until it matches the start of the current string.
* If the prefix becomes empty, return it immediately.
* Return the final prefix as the longest common prefix.

**Time Complexity:**

* **O(n),** where *n* is the total number of characters across all strings.

**Space Complexity:**

* **O(1),** using only a few variables.

**Code:**

def longestCommonPrefix(strs):

if not strs:

return ""

# Start with the first string as the initial common prefix

prefix = strs[0]

# Compare with each string in the array

for s in strs[1:]:

# Trim the prefix until it matches the start of the string

while not s.startswith(prefix):

prefix = prefix[:-1] # Reduce the prefix length

if prefix == "": # Early return if there's no common prefix

return ""

return prefix

**LeetCode** [**228: Summary Ranges**](https://leetcode.com/problems/summary-ranges/) 30/10/2024

You are given a sorted unique integer array nums.

A range [a,b] is the set of all integers from a to b (inclusive).

Return *the smallest sorted list of ranges that cover all the numbers in the array exactly*. That is, each element of nums is covered by exactly one of the ranges, and there is no integer x such that x is in one of the ranges but not in nums.

Each range [a,b] in the list should be output as:

* **"a->b" if a != b**
* **"a" if a == b**

**Approach**

1. Initialize an empty list ans to store the result ranges.
2. Use a pointer i to traverse through nums.
3. For each element in nums, mark it as start and then check consecutive elements to see if they form a sequence (i.e., if nums[i] + 1 == nums[i + 1]).
4. Move i forward until the sequence ends.
5. If start is different from the last element in the sequence (nums[i]), add the range as "start->nums[i]" to ans. If not, add start as a single value.
6. Repeat this until i reaches the end of nums.
7. Return the ans list containing the summary ranges.

**Time Complexity**

* **O(n)**, where n is the length of nums, as we iterate through nums once.

**Space Complexity**

* **O(n)**, for storing the result list ans.

**Code:**

class Solution(object):

def summaryRanges(self, nums):

ans = []

i = 0

while i < len(nums):

start = nums[i]

while i < len(nums)-1 and nums[i] + 1 == nums[i + 1]:

i += 1

if start != nums[i]:

ans.append(str(start) + "->" + str(nums[i]))

else:

ans.append(str(nums[i]))

i += 1

return ans

**LeetCode 238. Product of Array Except Self** 31/10/2024

Given an integer array nums, return *an array* answer *such that* answer[i] *is equal to the product of all the elements of* nums *except* nums[i].

The product of any prefix or suffix of nums is **guaranteed** to fit in a **32-bit** integer.

You must write an algorithm that runs in O(n) time and without using the division operation.

**Approach**

1. **Initialize Result Array**: Create an array result of size n (length of nums) initialized to 1.
2. **Prefix Products**: Traverse nums from left to right, maintaining a prefix product. For each index i, set result[i] to prefix and update prefix by multiplying it with nums[i].
3. **Suffix Products**: Traverse nums from right to left, maintaining a suffix product. For each index i, multiply result[i] by suffix and update suffix by multiplying it with nums[i].
4. The result array now holds the product of all elements except for the one at each index.

**Time Complexity**

* **O(n)**, where n is the length of nums, as we go through the array twice (once for prefixes and once for suffixes).

**Space Complexity**

* **O(1)** additional space, excluding the output array result (since no extra data structures are used other than the input and output arrays).

**Code:**

class Solution(object):

def productExceptSelf(self, nums):

n = len(nums)

result = [1] \* n

# Step 1: Calculate prefix products and store in result

prefix = 1

for i in range(n):

result[i] = prefix

prefix \*= nums[i]

# Step 2: Calculate suffix products and multiply with result

suffix = 1

for i in range(n - 1, -1, -1):

result[i] \*= suffix

suffix \*= nums[i]

return result

**LeetCode** [**56. Merge Intervals**](https://leetcode.com/problems/merge-intervals/) 01/11/2024

**Approach**

1. **Sort Intervals**: First, the intervals are sorted based on the starting value (interval[0]). This ensures that intervals are ordered from left to right.
2. **Merge Intervals**: Traverse through the sorted intervals:
   * If merged is empty or the current interval does not overlap with the last merged interval (merged[-1][1] < interval[0]), simply add the current interval to merged.
   * If there is an overlap (i.e., merged[-1][1] >= interval[0]), merge the intervals by updating the end of the last interval in merged to be the maximum of the two intervals' end values (max(merged[-1][1], interval[1])).
3. **Return Merged Intervals**: Finally, return the merged list, which contains the non-overlapping intervals after the merge.

**Time Complexity**

* **O(n log n)**, where n is the number of intervals. The sorting step takes **O(n log n)**, and the merging step takes **O(n)** (as we iterate through the list once).

**Space Complexity**

* **O(n)**, for the result list merged, which stores the merged intervals.

**Code:**

class Solution(object):

def merge(self, intervals):

intervals.sort(key=lambda interval: interval[0])

merged = []

for interval in intervals:

if not merged or merged[-1][1] < interval[0]:

merged.append(interval)

else:

merged[-1] = [merged[-1][0], max(merged[-1][1], interval[1])]

return merged

**LeetCode** [**54. Spiral Matrix**](https://leetcode.com/problems/merge-intervals/) 07/11/2024

Given an m x n matrix, return *all elements of the* matrix *in spiral order*.

**Approach**

1. Initialize the boundaries (left\_bound, right\_bound, up\_bound, down\_bound) for left, right, top, and bottom to control movement in the matrix.
2. Set the initial direction to **RIGHT** and initialize an empty list, result, to store the elements in spiral order.
3. Use a loop to traverse the matrix until all elements are added to result.
4. For each direction:
   * **RIGHT**: Traverse the row from left to right within bounds, then update row and boundaries, switching to the **DOWN** direction.
   * **DOWN**: Traverse the column from top to bottom within bounds, update row and boundaries, switching to **LEFT**.
   * **LEFT**: Traverse the row from right to left, update row and boundaries, switching to **UP**.
   * **UP**: Traverse the column from bottom to top, update row and boundaries, switching back to **RIGHT**.
5. Repeat until all elements are added to result.

**Time Complexity: O(m \* n)**

* **Explanation**: Each element in the matrix is accessed exactly once. Since there are m \* n elements, the complexity is **O(m \* n)**.

**Space Complexity: O(1) (ignoring output storage)**

* **Explanation**: The result list takes **O(m \* n)** space to store the output, but if we consider only auxiliary variables (ignoring the output list itself), then the space complexity is **O(1)** since no extra storage proportional to the input size is used.

**Code:**

def spiral\_matrix(matrix):  
 rows, cols = len(matrix), len(matrix[0])  
 i, j = 0, 0  
 LEFT, RIGHT, UP, DOWN = 0, 1, 2, 3  
 left\_bound, right\_bound, up\_bound, down\_bound = -1, cols, 0, rows  
 direction = RIGHT  
 result = []  
 while len(result) != rows \* cols:  
 if direction == RIGHT:  
 while j != right\_bound:  
 result.append(matrix[i][j])  
 j += 1  
 i, j = i + 1, j - 1  
 right\_bound -= 1  
 direction = DOWN  
 elif direction == DOWN:  
 while i != down\_bound:  
 result.append(matrix[i][j])  
 i += 1  
 i, j = i - 1, j - 1  
 down\_bound -= 1  
 direction = LEFT  
 elif direction == LEFT:  
 while j != left\_bound:  
 result.append(matrix[i][j])  
 j -= 1  
 i, j = i - 1, j + 1  
 left\_bound += 1  
 direction = UP  
 else: # direction == UP  
 while i != up\_bound:  
 result.append(matrix[i][j])  
 i -= 1  
 i, j = i + 1, j + 1  
 up\_bound += 1  
 direction = RIGHT  
 return result