***WEEK – 4***

**Problem 1. Group Anagrams**

Given an array of strings strs, group the anagrams together. You can return the answer in any order.

An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example 1:

Input: strs = ["eat","tea","tan","ate","nat","bat"]

Output: [["bat"],["nat","tan"],["ate","eat","tea"]]

Example 2:

Input: strs = [""]

Output: [[""]]

Example 3:

Input: strs = ["a"]

Output: [["a"]]

Constraints:

* 1 <= strs.length <= 104
* 0 <= strs[i].length <= 100
* strs[i] consists of lowercase English letters.

**Solution 1-**

class Solution:

def groupAnagrams(self, strs: List[str]) -> List[List[str]]:

anagrams = {}

for word in strs:

sortedWord = "".join(sorted(word))

if sortedWord in anagrams:

anagrams[sortedWord].append(word)

else:

anagrams[sortedWord] = [word]

return list(anagrams.values())

**Problem 2. Valid Anagram**

Given two strings s and t, return true *if* t *is an anagram of* s*, and* false *otherwise*.

An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example 1:

Input: s = "anagram", t = "nagaram"

Output: true

Example 2:

Input: s = "rat", t = "car"

Output: false

Constraints:

* 1 <= s.length, t.length <= 5 \* 104
* s and t consist of lowercase English letters.

**Solution 2-**

class Solution:

def isAnagram(self, s: str, t: str) -> bool:

return sorted(s) == sorted(t)

**Problem 3. Find Resultant Array After Removing Anagrams**

You are given a 0-indexed string array words, where words[i] consists of lowercase English letters.

In one operation, select any index i such that 0 < i < words.length and words[i - 1] and words[i] are anagrams, and delete words[i] from words. Keep performing this operation as long as you can select an index that satisfies the conditions.

Return words *after performing all operations*. It can be shown that selecting the indices for each operation in any arbitrary order will lead to the same result.

An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase using all the original letters exactly once. For example, "dacb" is an anagram of "abdc".

Example 1:

Input: words = ["abba","baba","bbaa","cd","cd"]

Output: ["abba","cd"]

Explanation:

One of the ways we can obtain the resultant array is by using the following operations:

- Since words[2] = "bbaa" and words[1] = "baba" are anagrams, we choose index 2 and delete words[2].

Now words = ["abba","baba","cd","cd"].

- Since words[1] = "baba" and words[0] = "abba" are anagrams, we choose index 1 and delete words[1].

Now words = ["abba","cd","cd"].

- Since words[2] = "cd" and words[1] = "cd" are anagrams, we choose index 2 and delete words[2].

Now words = ["abba","cd"].

We can no longer perform any operations, so ["abba","cd"] is the final answer.

Example 2:

Input: words = ["a","b","c","d","e"]

Output: ["a","b","c","d","e"]

Explanation:

No two adjacent strings in words are anagrams of each other, so no operations are performed.

Constraints:

* 1 <= words.length <= 100
* 1 <= words[i].length <= 10
* words[i] consists of lowercase English letters.

**Solution 3-**

class Solution:

def removeAnagrams(self, words: List[str]) -> List[str]:

i = 0

while i < len(words) - 1:

if sorted(words[i]) == sorted(words[i + 1]):

words.remove(words[i + 1])

continue

i += 1

return words

**Problem 4. Word Pattern**

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

Constraints:

* 1 <= pattern.length <= 300
* pattern contains only lower-case English letters.
* 1 <= s.length <= 3000
* s contains only lowercase English letters and spaces ' '.
* s does not contain any leading or trailing spaces.
* All the words in s are separated by a single space.

**Solution 4-**

class Solution:

def wordPattern(self, pattern: str, s: str) -> bool:

s\_list = s.split()

d = {}

pattern\_len = len(pattern)

if pattern\_len != len(s\_list):

return False

for i in range(pattern\_len):

if pattern[i] not in d:

if s\_list[i] in list(d.values()):

return False

d[pattern[i]] = s\_list[i]

else:

if d[pattern[i]] != s\_list[i]:

return False

return True

**Problem 5. Repeated Substring Pattern**

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

Example 1:

Input: s = "abab"

Output: true

Explanation: It is the substring "ab" twice.

Example 2:

Input: s = "aba"

Output: false

Example 3:

Input: s = "abcabcabcabc"

Output: true

Explanation: It is the substring "abc" four times or the substring "abcabc" twice.

Constraints:

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

**Solution 5-**

class Solution:

def repeatedSubstringPattern(self, s):

# if there is a k repeating pattern in s, then

# we can tell that a k-rotation of s should be equal to s

for k in range(1, len(s)//2 +1):

if s == s[k:] + s[:k]:

return True

return False

**Problem 6. Find and Replace Pattern**

Given a list of strings words and a string pattern, return *a list of* words[i] *that match* pattern. You may return the answer in any order.

A word matches the pattern if there exists a permutation of letters p so that after replacing every letter x in the pattern with p(x), we get the desired word.

Recall that a permutation of letters is a bijection from letters to letters: every letter maps to another letter, and no two letters map to the same letter.

Example 1:

Input: words = ["abc","deq","mee","aqq","dkd","ccc"], pattern = "abb"

Output: ["mee","aqq"]

Explanation: "mee" matches the pattern because there is a permutation {a -> m, b -> e, ...}.

"ccc" does not match the pattern because {a -> c, b -> c, ...} is not a permutation, since a and b map to the same letter.

Example 2:

Input: words = ["a","b","c"], pattern = "a"

Output: ["a","b","c"]

Constraints:

* 1 <= pattern.length <= 20
* 1 <= words.length <= 50
* words[i].length == pattern.length
* pattern and words[i] are lowercase English letters.

**Solution 6-**

class Solution:

def findAndReplacePattern(self, words: List[str], pattern: str) -> List[str]:

result = []

for word in words:

if len(set(pattern)) == len(set(word)):

tempDict = {}

Flag = False

for i in range(len(pattern)):

if pattern[i] not in tempDict:

Flag= True

tempDict[pattern[i]] = word[i]

elif pattern[i] in tempDict and tempDict[pattern[i]] != word[i]:

Flag = False

break

if Flag== True:

result.append(word)

return result

**Problem 7. Maximum Subarray**

Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.

A subarray is a contiguous part of an array.

Example 1:

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]

Output: 6

Explanation: [4,-1,2,1] has the largest sum = 6.

Example 2:

Input: nums = [1]

Output: 1

Example 3:

Input: nums = [5,4,-1,7,8]

Output: 23

Constraints:

* 1 <= nums.length <= 105
* -104 <= nums[i] <= 104

**Solution 7-**

class Solution:

def maxSubArray(self, nums: List[int]) -> int:

for i in range(1, len(nums)):

if nums[i - 1] > 0:

nums[i] += nums[i - 1]

return max(nums)

**Problem 8. Best Time to Buy and Sell Stock**

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.

Example 1:

Input: prices = [7,1,5,3,6,4]

Output: 5

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

Input: prices = [7,6,4,3,1]

Output: 0

Explanation: In this case, no transactions are done and the max profit = 0.

Constraints:

* 1 <= prices.length <= 105
* 0 <= prices[i] <= 104

**Solution 8-**

class Solution:

def maxProfit(self,prices):

left = 0 #Buy

right = 1 #Sell

max\_profit = 0

while right < len(prices):

currentProfit = prices[right] - prices[left] #our current Profit

if prices[left] < prices[right]:

max\_profit =max(currentProfit,max\_profit)

else:

left = right

right += 1

return max\_profit

**Problem 9. Best Time to Buy and Sell Stock II**

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

On each day, you may decide to buy and/or sell the stock. You can only hold at most one share of the stock at any time. However, you can buy it then immediately sell it on the same day.

Find and return *the maximum profit you can achieve*.

Example 1:

Input: prices = [7,1,5,3,6,4]

Output: 7

Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.

Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.

Total profit is 4 + 3 = 7.

Example 2:

Input: prices = [1,2,3,4,5]

Output: 4

Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.

Total profit is 4.

Example 3:

Input: prices = [7,6,4,3,1]

Output: 0

Explanation: There is no way to make a positive profit, so we never buy the stock to achieve the maximum profit of 0.

Constraints:

* 1 <= prices.length <= 3 \* 104
* 0 <= prices[i] <= 104

**Solution 9-**

class Solution:

def maxProfit(self, prices: List[int]) -> int:

running\_profit, max\_profit = 0, 0

for i in range(1, len(prices)):

running\_profit = max(0, prices[i] - prices[i-1])

max\_profit = max(max\_profit, max\_profit + running\_profit)

return max\_profit