ASSIGNMNET 24

1. 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.

```
def romanToInt(s):
    romanToNum = {
        'I': 1,
        'V': 5,
        'X': 10,
        'L': 50,
        'C': 100,
        'D': 500,
        'M': 1000
    }
    total = 0
    n = len(s)
    i = 0
    while i < n:
        if s[i] in romanToNum:
            if i + 1 < n and romanToNum[s[i + 1]] > romanToNum[s[i]]:
                total += romanToNum[s[i + 1]] - romanToNum[s[i]]
                i += 2
            else:
                total += romanToNum[s[i]]
                i += 1

    return total
```

2. Given a string `s`, find the length of the **longest substring** without repeating characters.

```
def lengthOfLongestSubstring(s):
    n = len(s)
    charIndexMap = {}
    maxLength = 0
    left = 0
    for right in range(n):
        if s[right] in charIndexMap:
            left = max(left, charIndexMap[s[right]] + 1)
        charIndexMap[s[right]] = right
        maxLength = max(maxLength, right - left + 1)

    return maxLength
```

3. Given an array `nums` of size `n`, return *the majority element*.

The majority element is the element that appears more than ⌊`n / 2`⌋ times. You may assume that the majority element always exists in the array.

```
def majorityElement(nums):
   candidate = None
   count = 0

   for num in nums:
      if count == 0:
         candidate = num
         count += 1
      elif num == candidate:
         count += 1
      else:
         count -= 1

   return candidate
```

4. 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.

```
from collections import defaultdict

def groupAnagrams(strs):
   anagramGroups = defaultdict(list)

   for s in strs:
      sortedS = ''.join(sorted(s))
      anagramGroups[sortedS].append(s)

   return list(anagramGroups.values())
```

5. An **ugly number** is a positive integer whose prime factors are limited to `2`, `3`, and `5`.

Given an integer `n`, return *the* `nth` ***ugly number***.

```
def nthUglyNumber(n):
   ugly = [0] * n
   ugly[0] = 1
   p2 = p3 = p5 = 0
   next_multiple_of_2 = 2
   next_multiple_of_3 = 3
   next_multiple_of_5 = 5
```

```
    for i in range(1, n):
        next_ugly = min(next_multiple_of_2, next_multiple_of_3, next_multiple_of_5)
        ugly[i] = next_ugly

        if next_ugly == next_multiple_of_2:
            p2 += 1
            next_multiple_of_2 = ugly[p2] * 2

        if next_ugly == next_multiple_of_3:
            p3 += 1
            next_multiple_of_3 = ugly[p3] * 3

        if next_ugly == next_multiple_of_5:
            p5 += 1
            next_multiple_of_5 = ugly[p5] * 5

    return ugly[n - 1]
```

6. Given an array of strings `words` and an integer `k`, return *the* `k` *most frequent strings*.

Return the answer **sorted** by **the frequency** from highest to lowest. Sort the words with the same frequency by their **lexicographical order**.

```
from collections import Counter

def topKFrequent(words, k):
    wordCount = Counter(words)

    def compareWords(word1, word2):
        if wordCount[word1] == wordCount[word2]:
            return -1 if word1 < word2 else 1
        return -1 if wordCount[word1] > wordCount[word2] else 1

    sortedWords = sorted(wordCount.keys(), key=compareWords)
    return sortedWords[:k]
```

7. You are given an array of integers `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position.

Return *the max sliding window*.

```
from collections import deque

def maxSlidingWindow(nums, k):
    window = deque()
    result = []
```

```
    for i in range(len(nums)):
        # Remove elements outside the left boundary of the sliding window
        if window and window[0] <= i - k:
            window.popleft()

        # Remove elements that are not potential maximums for future windows
        while window and nums[i] >= nums[window[-1]]:
            window.pop()

        # Add the current index to the window
        window.append(i)

        # Add the maximum element to the result
        if i >= k - 1:
            result.append(nums[window[0]])

    return result
```

8. Given a **sorted** integer array `arr`, two integers `k` and `x`, return the `k` closest integers to `x` in the array. The result should also be sorted in ascending order.

An integer `a` is closer to `x` than an integer `b` if:

- `|a - x| < |b - x|`, or
- `|a - x| == |b - x|` and `a < b`

```
def findClosestElements(arr, k, x):
    left = 0
    right = len(arr) - 1

    while right - left >= k:
        if abs(arr[left] - x) <= abs(arr[right] - x):
            right -= 1
        else:
            left += 1

    return arr[left:right + 1]
```