**DSA – ASSIGNMENT 24**

💡 1. **Roman to Integer**

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.

**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].

**Solution. :-**

* Define a dictionary symbolValues to store the mapping of Roman symbols to their corresponding values.
* Initialize a variable result to 0 to store the final integer value.
* Iterate through the Roman numeral string from left to right.
* For each symbol, do the following:
  + If the current symbol's value is less than the next symbol's value, subtract the current symbol's value from the result.
  + Otherwise, add the current symbol's value to the result.
* Return the final value stored in result.

**def romanToInt(s):**

**symbolValues = {**

**'I': 1,**

**'V': 5,**

**'X': 10,**

**'L': 50,**

**'C': 100,**

**'D': 500,**

**'M': 1000**

**}**

**result = 0**

**n = len(s)**

**for i in range(n):**

**if i < n - 1 and symbolValues[s[i]] < symbolValues[s[i+1]]:**

**result -= symbolValues[s[i]]**

**else:**

**result += symbolValues[s[i]]**

**return result**

**roman\_numeral = "III"**

**print(romanToInt(roman\_numeral))**

💡 2. **Longest Substring Without Repeating Characters**

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

**Example 1:**

Input: s = "abcabcbb"

Output: 3

Explanation: The answer is "abc", with the length of 3.

**Example 2:**

Input: s = "bbbbb"

Output: 1

Explanation: The answer is "b", with the length of 1.

**Example 3:**

Input: s = "pwwkew"

Output: 3

Explanation: The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

**Constraints:**

* 0 <= s.length <= 50000
* s consists of English letters, digits, symbols and spaces.

**Solution. :-**

* Initialize two pointers, start and end, to keep track of the current substring.
* Initialize a set, unique\_chars, to store the unique characters in the current substring.
* Initialize a variable, max\_length, to store the length of the longest substring without repeating characters.
* Iterate the end pointer through the string s, from left to right:
  + If the character at end is not in unique\_chars, add it to the set and update max\_length if necessary.
  + If the character at end is already in unique\_chars, remove the character at start from unique\_chars and increment start to move the sliding window.
* Return the max\_length as the length of the longest substring without repeating characters.

**def lengthOfLongestSubstring(s):**

**n = len(s)**

**unique\_chars = set()**

**start = 0**

**end = 0**

**max\_length = 0**

**while start < n and end < n:**

**if s[end] not in unique\_chars:**

**unique\_chars.add(s[end])**

**end += 1**

**max\_length = max(max\_length, end - start)**

**else:**

**unique\_chars.remove(s[start])**

**start += 1**

**return max\_length**

**string = "abcabcbb"**

**print(lengthOfLongestSubstring(string))**

💡 3. **Majority Element**

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.

**Example 1:**

Input: nums = [3,2,3]

Output: 3

**Example 2:**

Input: nums = [2,2,1,1,1,2,2]

Output: 2

**Constraints:**

* n == nums.length
* 1 <= n <= 5 \* 10^4
* -10^9 <= nums[i] <= 10^9

**Solution. :-**

* Initialize two variables, majority\_element and count, to keep track of the current majority element and its count.
* Iterate through the array nums:
  + If the count is 0, assign the current element as the majority\_element and set count to 1.
  + If the current element is equal to the majority\_element, increment count by 1.
  + If the current element is different from the majority\_element, decrement count by 1.
* Return the majority\_element as the majority element in the array.

**def majorityElement(nums):**

**majority\_element = None**

**count = 0**

**for num in nums:**

**if count == 0:**

**majority\_element = num**

**count = 1**

**elif num == majority\_element:**

**count += 1**

**else:**

**count -= 1**

**return majority\_element**

**nums = [3, 2, 3]**

**print(majorityElement(nums))**

💡 4. **Group Anagram**

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 <= 10000
* 0 <= strs[i].length <= 100
* strs[i] consists of lowercase English letters.

**Solution. :-**

* Initialize an empty dictionary anagram\_groups to store the groups of anagrams.
* Iterate through each string s in the input array strs:
  + Sort the characters of s and use it as a key to access the corresponding group in anagram\_groups.
  + If the key doesn't exist in anagram\_groups, create a new list for the key and add s to the list.
  + If the key already exists, append s to the list associated with the key.
* Return the values of anagram\_groups as the grouped anagrams.

**def groupAnagrams(strs):**

**anagram\_groups = {}**

**for s in strs:**

**sorted\_s = ''.join(sorted(s))**

**if sorted\_s not in anagram\_groups:**

**anagram\_groups[sorted\_s] = [s]**

**else:**

**anagram\_groups[sorted\_s].append(s)**

**return list(anagram\_groups.values())**

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

**print(groupAnagrams(strs))**

💡 5. **Ugly Numbers**

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***.

**Example 1:**

Input: n = 10

Output: 12

Explanation: [1, 2, 3, 4, 5, 6, 8, 9, 10, 12] is the sequence of the first 10 ugly numbers.

**Example 2:**

Input: n = 1

Output: 1

Explanation: 1 has no prime factors, therefore all of its prime factors are limited to 2, 3, and 5.

**Constraints:**

* 1 <= n <= 1690

**Solution. :-**

* Create an array ugly of size n to store the ugly numbers.
* Initialize three pointers p2, p3, and p5 to track the next multiple of 2, 3, and 5 respectively. Set their initial values to 0.
* Set the first ugly number ugly[0] to 1.
* Iterate from i = 1 to n-1:
  + Calculate the next ugly number as the minimum of ugly[p2]\*2, ugly[p3]\*3, and ugly[p5]\*5.
  + Update the pointers p2, p3, and p5 based on the calculated next ugly number:
    - If ugly[p2]\*2 is equal to the next ugly number, increment p2 by 1.
    - If ugly[p3]\*3 is equal to the next ugly number, increment p3 by 1.
    - If ugly[p5]\*5 is equal to the next ugly number, increment p5 by 1.
  + Assign the calculated next ugly number to ugly[i].
* Return the last element of the ugly array as the nth ugly number.

**def nthUglyNumber(n):**

**ugly = [0] \* n**

**ugly[0] = 1**

**p2 = p3 = p5 = 0**

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

**next\_ugly = min(ugly[p2] \* 2, ugly[p3] \* 3, ugly[p5] \* 5)**

**ugly[i] = next\_ugly**

**if next\_ugly == ugly[p2] \* 2:**

**p2 += 1**

**if next\_ugly == ugly[p3] \* 3:**

**p3 += 1**

**if next\_ugly == ugly[p5] \* 5:**

**p5 += 1**

**return ugly[n-1]**

**n = 10**

**print(nthUglyNumber(n))**

💡 6. **Top K Frequent Words**

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**.

**Example 1:**

Input: words = ["i","love","leetcode","i","love","coding"], k = 2

Output: ["i","love"]

Explanation: "i" and "love" are the two most frequent words.

Note that "i" comes before "love" due to a lower alphabetical order.

**Example 2:**

Input: words = ["the","day","is","sunny","the","the","the","sunny","is","is"], k = 4

Output: ["the","is","sunny","day"]

Explanation: "the", "is", "sunny" and "day" are the four most frequent words, with the number of occurrence being 4, 3, 2 and 1 respectively.

**Constraints:**

* 1 <= words.length <= 500
* 1 <= words[i].length <= 10
* words[i] consists of lowercase English letters.
* k is in the range [1, The number of \*\*unique\*\* words[i]]

**Solution. :-**

* Create a dictionary word\_count to store the count of each word.
* Iterate over each word in the input words array:
  + If the word is already in word\_count, increment its count by 1.
  + Otherwise, add the word to word\_count with an initial count of 1.
* Create an empty list freq\_heap to store the words and their counts as tuples.
* Iterate over each key-value pair in word\_count:
  + Add a tuple (count, word) to freq\_heap, where count is the negative value of the word count to create a max heap based on the counts. In case of ties in counts, use the lexicographical order of the words.
* Use the heapq module to convert the freq\_heap list into a max heap.
* Create an empty list result to store the k most frequent words.
* Pop the top k elements from the max heap and append the word part of each tuple to result.
* Return the result list.

**import heapq**

**from collections import Counter**

**def topKFrequent(words, k):**

**word\_count = Counter(words)**

**freq\_heap = []**

**for word, count in word\_count.items():**

**heapq.heappush(freq\_heap, (-count, word))**

**result = []**

**for \_ in range(k):**

**result.append(heapq.heappop(freq\_heap)[1])**

**return result**

**words = ["i", "love", "leetcode", "i", "love", "coding"]**

**k = 2**

**print(topKFrequent(words, k))**

💡 7. **Sliding Window Maximum**

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*.

**Example 1:**

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3

Output: [3,3,5,5,6,7]

Explanation:

Window position Max

--------------- -----

[1 3 -1] -3 5 3 6 7 3

1 [3 -1 -3] 5 3 6 7 3

1 3 [-1 -3 5] 3 6 7 5

1 3 -1 [-3 5 3] 6 7 5

1 3 -1 -3 [5 3 6]7 6

1 3 -1 -3 5 [3 6 7] 7

**Example 2:**

Input: nums = [1], k = 1

Output: [1]

**Constraints:**

* 1 <= nums.length <= 100000
* -10000 <= nums[i] <= 10000
* 1 <= k <= nums.length

**Solution. :-**

* Create an empty deque window to store the indices of elements in the sliding window.
* Create an empty list result to store the maximum values for each sliding window.
* Iterate over each index i in the input nums array:
  + If the leftmost index in window is outside the current sliding window, remove it from the left of window.
  + While window is not empty and the element at the rightmost index in window is less than or equal to the current element at index i, remove it from the right of window.
  + Append the current index i to window.
  + If the current index i is greater than or equal to k - 1, it means the first sliding window is complete. Append the maximum element in the sliding window (which is the element at the leftmost index in window) to result.
* Return the result list.

**from collections import deque**

**def maxSlidingWindow(nums, k):**

**window = deque()**

**result = []**

**for i in range(len(nums)):**

**if window and window[0] <= i - k:**

**window.popleft()**

**while window and nums[window[-1]] <= nums[i]:**

**window.pop()**

**window.append(i)**

**if i >= k - 1:**

**result.append(nums[window[0]])**

**return result**

**nums = [1, 3, -1, -3, 5, 3, 6, 7]**

**k = 3**

**print(maxSlidingWindow(nums, k))**

💡 8. **Find K Closest Elements**

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

**Example 1:**

Input: arr = [1,2,3,4,5], k = 4, x = 3

Output: [1,2,3,4]

**Example 2:**

Input: arr = [1,2,3,4,5], k = 4, x = -1

Output: [1,2,3,4]

**Constraints:**

* 1 <= k <= arr.length
* 1 <= arr.length <= 10000
* arr is sorted in **ascending** order.
* -10000 <= arr[i], x <= 10000

**Solution. :-**

* Perform a binary search on the arr to find the index left of the element that is less than or equal to x.
  + Set left to 0 and right to len(arr) - 1.
  + While left is less than right, calculate the mid index as mid = (left + right) // 2.
    - If the element at index mid is greater than x, set right = mid - 1.
    - Otherwise, set left = mid + 1.
  + After the binary search, left will be the index of the element that is less than or equal to x.
* Initialize two pointers low and high as left and left + 1.
* While k is greater than 0, do the following:
  + If low is greater than or equal to 0 and high is less than len(arr):
    - If |arr[low] - x| is less than or equal to |arr[high] - x|:
      * Append arr[low] to the result list.
      * Decrement low by 1.
    - Otherwise:
      * Append arr[high] to the result list.
      * Increment high by 1.
  + If low is less than 0 and high is less than len(arr):
    - Append arr[high] to the result list.
    - Increment high by 1.
  + If high is greater than or equal to len(arr) and low is greater than or equal to 0:
    - Append arr[low] to the result list.
    - Decrement low by 1.
  + Decrement k by 1.
* Return the result list.

**def findClosestElements(arr, k, x):**

**left = 0**

**right = len(arr) - 1**

**while left < right:**

**mid = (left + right) // 2**

**if arr[mid] > x:**

**right = mid - 1**

**else:**

**left = mid + 1**

**low = left - 1**

**high = left**

**result = []**

**while k > 0:**

**if low >= 0 and high < len(arr):**

**if abs(arr[low] - x) <= abs(arr[high] - x):**

**result.append(arr[low])**

**low -= 1**

**else:**

**result.append(arr[high])**

**high += 1**

**elif low >= 0:**

**result.append(arr[low])**

**low -= 1**

**elif high < len(arr):**

**result.append(arr[high])**

**high += 1**

**k -= 1**

**return sorted(result)**

**arr = [1, 2, 3, 4, 5]**

**k = 4**

**x = 3**

**print(findClosestElements(arr, k, x))**