1. **Roman to Integer**

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.

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:

```
def roman_to_int(s):
    roman_values = {
        'I': 1,
        'V': 5,
        'X': 10,
        'L': 50,
        'C': 100,
        'D': 500,
        'M': 1000
    }

    # Initialize the result
    result = 0

    # Iterate through the characters of the Roman numeral
    for i in range(len(s)):
        # Get the integer value of the current character
        value = roman_values[s[i]]

        # If the current character is followed by a larger value character, subtract the current value
```

```
        if i < len(s) - 1 and roman_values[s[i+1]] > value:
            result -= value
        else:
            result += value

    return result

# Example usage
s = "LVIII"
print("Roman:", s)
print("Integer:", roman_to_int(s))
```

---

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:

```
def length_of_longest_substring(s):
    # Initialize variables
    max_length = 0
    start = 0
    seen = {}

    # Iterate through the characters of the string
    for end in range(len(s)):
        # Check if the current character is already seen in the current window
        if s[end] in seen and start <= seen[s[end]]:
            # Update the start index to the next position after the repeating character
            start = seen[s[end]] + 1

        # Update the maximum length
        max_length = max(max_length, end - start + 1)

        # Store the index of the current character
        seen[s[end]] = end

    return max_length
```

---

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:

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

    # Find the candidate for the majority element
    for num in nums:
        if count == 0:
            candidate = num
        if num == candidate:
            count += 1
        else:
            count -= 1

    return candidate
```

---

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:

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

    # Find the candidate for the majority element
    for num in nums:
        if count == 0:
            candidate = num
        if num == candidate:
            count += 1
        else:
            count -= 1

    return candidate
```

---

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:

```
from collections import defaultdict


def group_anagrams(strs):
    anagram_groups = defaultdict(list)
```

```
    # Group the anagrams by sorting the characters
    for word in strs:
        sorted_word = ''.join(sorted(word))
        anagram_groups[sorted_word].append(word)

    # Convert the values of the hash table into a list of groups
    result = list(anagram_groups.values())

    return result
```

---

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:

```
def nth_ugly_number(n):
    ugly_numbers = [1]  # Store the ugly numbers
    p2 = p3 = p5 = 0  # Pointers to track the indices for multiplying by 2, 3, and 5

    # Generate the subsequent ugly numbers
    for _ in range(1, n):
        # Compute the next ugly number by multiplying the existing ugly numbers with 2, 3, and 5
        next_ugly = min(ugly_numbers[p2] * 2, ugly_numbers[p3] * 3, ugly_numbers[p5] * 5)

        # Update the pointers based on the next ugly number
        if next_ugly == ugly_numbers[p2] * 2:
            p2 += 1
        if next_ugly == ugly_numbers[p3] * 3:
            p3 += 1
        if next_ugly == ugly_numbers[p5] * 5:
            p5 += 1

        # Add the next ugly number to the list
        ugly_numbers.append(next_ugly)
```

return ugly_numbers[-1]

---

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]]` </aside>

Solution:

def topKFrequent(words, k):

```
    # Step 1: Create a dictionary to store the frequency of each word
    frequency_dict = {}
    for word in words:
        frequency_dict[word] = frequency_dict.get(word, 0) + 1

    # Step 2: Sort the words based on frequency and lexicographical order
    sorted_words = sorted(frequency_dict.keys(), key=lambda x: (-frequency_dict[x], x))

    # Step 3: Extract the top k frequent words
    result = sorted_words[:k]

    return result
```

---

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` </aside>

Solution:

```
from collections import deque


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

    for i in range(len(nums)):
        # Remove elements outside the current sliding window from the front of the deque
        if deque and deque[0] == i - k:
```

```
        deque.popleft()


    # Remove elements smaller than the current element from the back of the deque
    while deque and nums[deque[-1]] < nums[i]:
        deque.pop()


    deque.append(i)


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


return result
```

---

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:

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

    # Binary search to find the position of x
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == x:
            targetIndex = mid
            break
        elif arr[mid] < x:
            left = mid + 1
        else:
            right = mid - 1
```

```
else:
    # x is not found, so the target index is the left pointer
    targetIndex = left

# Initialize the left and right pointers for finding k closest elements
left = targetIndex - 1
right = targetIndex

# Expand towards the left and right to find k closest elements
while right - left - 1 < k:
    if left < 0:
        right += 1
    elif right >= len(arr):
        left -= 1
    elif abs(x - arr[left]) <= abs(x - arr[right]):
        left -= 1
    else:
        right += 1

# Return the subarray of k closest elements
return arr[left + 1:right]
```