4. Median of Two Sorted Arrays

For this question, we only need to find a position i in the first array and second position j in the second array, that meets the requirement nums1[i - 1] < nums2[j] and nums1[j - 1] < nums2[i] and i + j == (len(nums1) + len(nums2) + 1) // 2. So in the end, when the total length is odd, the median is max(nums1[i - 1], nums2[j - 1]), when the length is even, the median should be avg(max(nums1[i - 1], nums2[j - 1]), min(nums1[i], nums2[j])). Also we need to consider edge cases. The time complexity is log(min(m,n))

```
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:
        m, n = len(nums1), len(nums2)
        if m > n:
            return self.findMedianSortedArrays(nums2, nums1)
        imin, imax, half_len = 0, m, (m + n + 1) // 2
        while imin <= imax:
            i = (imin + imax) // 2
            j = half_len - i
            if i > 0 and nums1[i - 1] > nums2[j]:
                imax = i - 1
            elif i < m and nums1[i] < nums2[j - 1]:
                imin = i + 1
            else:
                if i == 0:
                    max_left = nums2[j - 1]
                elif j == 0:
                    max_left = nums1[i - 1]
                else:
                    max_left = max(nums2[j - 1], nums1[i - 1])
                if (m + n) % 2 == 1:
                    return max_left

                if i == m:
                    min_right = nums2[j]
                elif j == n:
                    min_right = nums1[i]
                else:
                    min_right = min(nums2[j], nums1[i])
                return (max_left + min_right) / 2.0
```

200. Number of islands

The basic way is to iterate through all cells and when encounter unvisited ‘1’, then use bfs to traverse all adjacent islands to make them visited. Before each bfs, we will increase our count by 1. The TC is O(rows * cols)

```
from collections import deque
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        if not grid or not grid[0]:
            return 0
        visited = {}
        directions = [[0, 1], [0, -1], [1, 0], [-1, 0]]
        q = deque()
        count = 0
        rows, cols = len(grid), len(grid[0])
        for i in range(rows):
            for j in range(cols):
                if grid[i][j] == '1' and (i, j) not in visited:
                    count += 1
                    q.clear()
                    q.append((i, j))
                    visited[(i, j)] = True
                    while len(q) > 0:
                        x, y = q.popleft()
                        for d_x, d_y in directions:
                            if 0 <= x + d_x < rows and 0 <= y + d_y < cols and grid[x + d_x][y + d_y] == '1'
and (x + d_x, y + d_y) not in visited:
                                q.append((x + d_x, y + d_y))
                                visited[(x + d_x, y + d_y)] = True
        return count
```

42. Trapping Rain Water

In this question, we will iterate from both sides, we only need to memorize the left highest height and right highest height. We will move the side which has the highest side so that we could confirm that it could trap that height of water. We calculate on column by one until to indexes meet. The TC is O(n).

```
 class Solution:
    def trap(self, height: List[int]) -> int:
        if not height:
            return 0
        left, right = 0, len(height) - 1
        result = 0
        left_h, right_h = height[0], height[-1]
        while left <= right:
            if left_h < right_h:
                if left_h < height[left]:
                    left_h = height[left]
```

```
            elif left_h > height[left]:
                result += left_h - height[left]
            left += 1
        else:
            if right_h < height[right]:
                right_h = height[right]
            elif right_h > height[right]:
                result += right_h - height[right]
            right -= 1
    return result
```

15. 3Sum

For this question, we need to sort the array and iterate through all elements. For each element, we need to find other two elements from two sides. That downgrades to two-sum question. We will go through the next element if it's the same as this one to prevent the duplication. Also when we find other two elements, we also need to skip duplicate elements. The TC is O(n**2)

```
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        nums.sort()
        result = []
        prev = None
        length = len(nums)
        for i in range(length - 2):
            if prev == nums[i]:
                continue
            else:
                prev = nums[i]
            left, right = i + 1, length - 1
            target = -nums[i]
            while left < right:
                if nums[left] + nums[right] < target:
                    left += 1
                elif nums[left] + nums[right] > target:
                    right -= 1
                else:
                    result.append([nums[i], nums[left], nums[right]])
                    left += 1
                    while left < length and nums[left - 1] == nums[left]:
                        left += 1
                    right -= 1
                    while right >= 0 and nums[right + 1] == nums[right]:
                        right -= 1
```

```
        return result
```

3. Longest Substring Without Repeating Characters

For this kind of question, we could use slidewindow. We could use a map to memorize every letter's presenting time. Once it's more than 0, it means we should increase left bound to get rid of this duplication. At that time we should compare the length of longest substring with the stored one. And always store the maximum one. In the end, we will return the max(cur, store) as the final result.

```
from collections import defaultdict
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        max_length = 0
        memo = defaultdict(int)
        length = len(s)
        count = 0
        left = 0
        for i in range(length):
            if memo[s[i]] == 0:
                memo[s[i]] += 1
                count += 1
            else:
                if max_length < count:
                    max_length = count
                while s[left] != s[i]:
                    memo[s[left]] -= 1
                    count -= 1
                    left += 1
                left += 1
        return max(max_length, count)
```