TIP103 - Mock Technical Interview Questions

Median of Two Sorted Arrays (Hard)
Merge k Sorted Lists (Hard)

Median of Two Sorted Arrays

Problem:

Given two sorted arrays `nums1` and `nums2` of size `m` and `n` respectively, return **the median** of the two sorted arrays.

The overall run time complexity should be `O(log (m+n))`.

Example 1:

```
Input: nums1 = [1,3], nums2 = [2]
Output: 2.00000
Explanation: merged array = [1,2,3] and median is 2.
```

Example 2:

```
Input: nums1 = [1,2], nums2 = [3,4]
Output: 2.50000
Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.
```

Clarifying Questions

- Are the input arrays guaranteed to be sorted?
  - Yes
- Can the input arrays have different lengths?
  - Yes
- Can the input arrays be empty?
  - No
- What should be returned if one of the input arrays is empty?
  - If one of the input arrays is empty, the median of the other array should be returned
- Is it guaranteed that the combined length of the input arrays is odd or even?

- The combined length of the input arrays can be odd or even

Hints

- The **median of an array** is the middle element if the array has odd length, or the average of the two middle elements if the array has even length.
- To find the median of two sorted arrays, we can **combine the arrays** into a single sorted array and then **find the median of the combined array**.
- To combine the arrays, we can use a **two-pointer approach** where we compare the elements at the current pointers in the two arrays and add the smaller element to the combined array.
- To find the median of the combined array, we can use the **same approach** as finding the **median of a single array**.

General Approach

1. Check if one of the input arrays is empty. If so, return the median of the non-empty array.
2. Initialize two pointers `i` and `j` to the start of the two input arrays, respectively.
3. Initialize an empty array `combined` to store the combined array.
4. While the pointers `i` and `j` are within the bounds of the input arrays:
    1. Compare the elements at the current pointers in the two input arrays.
    2. Add the smaller element to the `combined` array.
    3. Move the pointer of the array with the smaller element to the next element.
5. If one of the pointers is still within the bounds of its array, add the remaining elements of the array to the `combined` array.
6. If the combined length is odd, return the middle element of the `combined` array.
7. If the combined length is even, return the average of the two middle elements of the `combined` array.

Space and Time Complexity

- The *space complexity* of this approach is **O(m+n)**, where m and n are the lengths of the input arrays. This is because we need to store the combined array in memory.
- The *time complexity* of this approach is **O(m+n)**, where m and n are the lengths of the input arrays. This is because we need to loop through all elements of the input arrays to combine them and find the median.

SOLUTION

Python

```
class Solution:
    def findMedianSortedArrays(self, nums1, nums2):
        # Check if one of the input arrays is empty
        if not nums1:
            return self.findMedian(nums2)
        if not nums2:
            return self.findMedian(nums1)

        # Initialize pointers to the start of the input arrays
        i = 0
        j = 0

        # Initialize empty list to store the combined array
        combined = []

        # Combine the input arrays
        while i < len(nums1) and j < len(nums2):
            if nums1[i] <= nums2[j]:
                combined.append(nums1[i])
                i += 1
            else:
                combined.append(nums2[j])
                j += 1

        # Add remaining elements of the input arrays to the combined array
        combined += nums1[i:]
        combined += nums2[j:]

        # Return the median of the combined array
        return self.findMedian(combined)

    def findMedian(self, nums):
        # Check if the array has odd length
        if len(nums) % 2 == 1:
            return nums[len(nums) // 2]

        # Return the average of the two middle elements
        return (nums[len(nums) // 2 - 1] + nums[len(nums) // 2]) / 2
```

Java

```
    class Solution {
        public double findMedianSortedArrays(int[] nums1, int[] nums2) {
            // Check if one of the input arrays is empty
            if (nums1.length == 0) {
                return findMedian(nums2);
            }
            if (nums2.length == 0) {
                return findMedian(nums1);
            }

            // Initialize pointers to the start of the input arrays
            int i = 0;
            int j = 0;

            // Initialize empty list to store the combined array
            List<Integer> combined = new ArrayList<>();

            // Combine the input arrays
            while (i < nums1.length && j < nums2.length) {
                if (nums1[i] <= nums2[j]) {
                    combined.add(nums1[i]);
                    i++;
                } else {
                    combined.add(nums2[j]);
                    j++;
                }
            }

            // Add remaining elements of the input arrays to the combined array
            for (int k = i; k < nums1.length; k++) {
                combined.add(nums1[k]);
            }
            for (int k = j; k < nums2.length; k++) {
                combined.add(nums2[k]);
            }

            // Convert the combined list to an array
            int[] combinedArray = combined.stream().mapToInt(Integer::intValue).toArra

            // Return the median of the combined array
            return findMedian(combinedArray);
        }

        private double findMedian(int[] nums) {
            // Check if the array has odd length
            if (nums.length % 2 == 1) {
                return nums[nums.length / 2];
            }
```

```
            // Return the average of the two middle elements
            return (nums[nums.length / 2 - 1] + nums[nums.length / 2]) / 2.0;
        }
    }
```

Follow-up Questions

- How would you extend your solution to handle finding the median of more than two sorted arrays?
- Can you provide an example input where your solution would not work and explain why?
- Can you think of any alternative approaches to solving the problem?
- Can you think of any real-world scenarios where the problem could be applied?

Leetcode Reference

Median of Two Sorted Arrays (https://leetcode.com/problems/median-of-two-sorted-arrays/)

Merge k Sorted Lists

Problem:

You are given an array of `k` linked-lists `lists`, each linked-list is sorted in ascending order.

Merge all the linked-lists into one sorted linked-list and return it.

Example 1:

```
Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]
Explanation: The linked-lists are:
[
  1->4->5,
  1->3->4,
  2->6
]
merging them into one sorted list:
1->1->2->3->4->4->5->6
```

Example 2:

```
Input: lists = []
Output: []
```

Example 3:

```
Input: lists = [[]]
Output: []
```

Clarifying Questions

- Are the input lists guaranteed to be sorted in ascending order?
  - Yes
- Can the input lists have different lengths?
  - Yes
- Can the input lists be empty?
  - Yes
- What should be returned if the input lists are empty?
  - If the input lists are empty, the function should return `None` or `null`

Hints

- To merge multiple sorted lists into a single sorted list, we can use a **two-pointer approach** where we compare the elements at the current pointers in the lists and add the smallest element to the resulting list.
- To find the smallest element, we can **use a priority queue** that stores the elements at the current pointers of the input lists. This will allow us to efficiently get the smallest element and move the pointer of the corresponding list to the next element.
- **After merging** the input lists, we can **return the resulting list** as the answer.

General Approach

1. Check if the input lists are empty. If so, return an empty list.
2. Initialize a priority queue that compares the elements of the input lists.
3. Initialize an empty list `result` to store the resulting list.
4. While the priority queue is not empty:
5. Remove the smallest element from the priority queue and add it to the `result` list.
6. If the list that the element came from has more elements, add the next element from that list to the priority queue.
7. Return the `result` list.

Space and Time Complexity

The *time complexity* of this approach is **O(n * log k)**, where n is the total number of nodes in the input lists and k is the number of lists. This is because the time complexity of the mergeTwoLists function is O(n) and it is called k * log k times in the mergeKLists function. The *space complexity* of the code is **O(n)**, where n is the total number of nodes in all of the lists. This is because the priority queue will store at most n nodes at any given time...

SOLUTION

Python

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def mergeKLists(self, lists: List[ListNode]) -> ListNode:
        # If there are no lists, return None
        if not lists:
            return None

        # If there is only one list, return that list
        if len(lists) == 1:
            return lists[0]

        # If there are only two lists, return their merge
        if len(lists) == 2:
            return self.mergeTwoLists(lists[0], lists[1])

        # Recursively merge the lists in pairs
        mid = len(lists) // 2
        left = self.mergeKLists(lists[:mid])
        right = self.mergeKLists(lists[mid:])

        # Return the merged lists
        return self.mergeTwoLists(left, right)

    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        # If either of the lists is empty, return the other list
        if not l1:
            return l2
        if not l2:
            return l1

        # Initialize the result list
        result = ListNode(0)
        current = result

        # Loop until one of the lists is empty
        while l1 and l2:
            # If the value of the first list is smaller, add it to the result and
            if l1.val < l2.val:
                current.next = ListNode(l1.val)
                l1 = l1.next
            # If the value of the second list is smaller, add it to the result and
            else:
                current.next = ListNode(l2.val)
```

```
            l2 = l2.next
        # Move to the next node in the result list
        current = current.next

    # Append the remaining nodes in the non-empty list
    current.next = l1 if l1 else l2

    # Return the merged list
    return result.next
```

Java

```
public class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        // If the list of lists is null or empty, return null
        if (lists == null || lists.length == 0) return null;

        // Create a priority queue to store the nodes of the lists in order
        PriorityQueue<ListNode> queue= new PriorityQueue<ListNode>(lists.length,ne
            @Override
            public int compare(ListNode l1, ListNode l2){
                if (l1.val < l2.val)
                    return -1;
                else if (l1.val == l2.val)
                    return 0;
                else
                    return 1;
            }
        });

        // Initialize the result list
        ListNode result = new ListNode(0);
        ListNode current = result;

        // Add the non-null nodes of the lists to the priority queue
        for (ListNode node:lists)
            if (node != null)
                queue.add(node);

        // Loop until the queue is empty
        while (!queue.isEmpty()){
            // Get the first node in the queue and add it to the result list
            current.next = queue.poll();
            current = current.next;

            // If the current node has a next node, add it to the queue
            if (current.next != null)
                queue.add(current.next);
        }

        // Return the result list
        return result.next;
    }
}
```

Follow-up Questions

- How would you improve the performance of your solution?

- Can you provide an example input where your solution would not work and explain why?
- Can you think of any alternative approaches to solving the problem?
- Can you think of any real-world scenarios where the problem could be applied?

Leetcode Reference

Merge k Sorted Lists (https://leetcode.com/problems/merge-k-sorted-lists/)